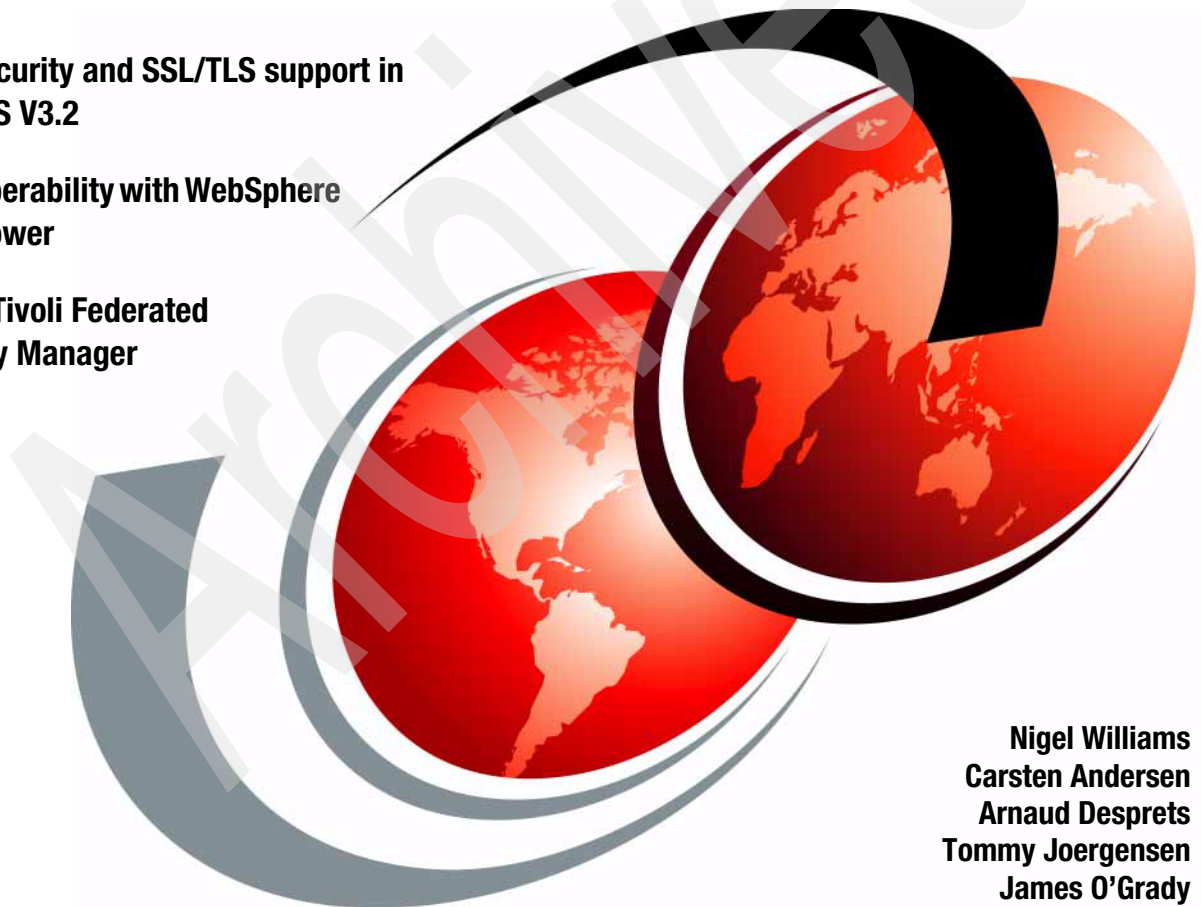


Securing CICS Web Services

WS-Security and SSL/TLS support in
CICS TS V3.2

Interoperability with WebSphere
DataPower

Using Tivoli Federated
Identity Manager



Nigel Williams
Carsten Andersen
Arnaud Desprets
Tommy Joergensen
James O'Grady



International Technical Support Organization

Securing CICS Web Services

November 2008

Archived

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

First Edition (November 2008)

This edition applies to Version 3, Release 2, CICS Transaction Server.

© Copyright International Business Machines Corporation 2008. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix
Trademarks	x
Preface	xi
The team that wrote this book	xii
Become a published author	xiii
Comments welcome	xiv
Chapter 1. Security for CICS Web services	1
1.1 Traditional CICS security	2
1.1.1 CICS user IDs	2
1.1.2 Special CICS user IDs	3
1.2 Security exposures	3
1.3 Security objectives	5
1.4 CICS resources used for securing Web services	7
1.4.1 CICS as a service provider	7
1.4.2 CICS as a service requester	13
1.5 Transport security	16
1.5.1 HTTP transport	16
1.5.2 WebSphere MQ transport	25
Chapter 2. SOAP message security	31
2.1 WS-Security	32
2.1.1 Web Services-Security road map	34
2.1.2 Web Services-Security model framework	34
2.2 CICS and SOAP message security	35
2.2.1 CICS support for WS-Trust	37
2.2.2 SOAP message security options	38
2.2.3 Enabling CICS for WS-Security processing	47
2.2.4 Configuring the CICS-supplied security handler	48
2.2.5 Custom security handlers	53
2.3 WebSphere and SOAP message security	55
2.4 Identity assertion	58
2.4.1 Trust token model	59
2.4.2 Blind trust model	59
2.5 WebSphere DataPower SOA appliances	59
2.6 Comparison of transport level and SOAP message security	61
2.7 Planning considerations	63

Chapter 3. Elements of cryptography	69
3.1 The role of cryptography	70
3.2 Secret key (or symmetric) cryptography	70
3.2.1 DES	72
3.2.2 Triple DES (TDEA)	77
3.2.3 AES	78
3.3 Public key (or asymmetric) cryptography	79
3.3.1 RSA	80
3.3.2 Digital envelopes	83
3.3.3 Appropriate key size in the RSA cryptosystem	84
3.4 Hash functions	84
3.5 Message authentication codes	88
3.5.1 Block cipher-based MACs	88
3.5.2 Hash function-based MACs	89
3.6 Digital signatures	90
3.6.1 Using DSA for digital signatures	91
3.6.2 Using RSA for digital signatures	93
3.6.3 Comparing RSA with DSA for digital signatures	96
3.7 Public key digital certificates	97
3.7.1 tbsCertificate	99
3.7.2 Standard extensions for X.509 V3 digital certificates	104
3.7.3 Certification paths	107
3.8 Certificate revocation lists	109
3.8.1 Extensions for entries in a CRL	111
3.8.2 Extensions for a CRL	112
3.8.3 Security considerations when using digital certificates	112
3.9 Key agreement protocols	113
3.9.1 The RSA key agreement protocol	114
3.9.2 The Diffie-Hellman key agreement protocol	114
3.10 Transport Layer Security (TLS) 1.0 protocol	116
3.10.1 TLS overview	117
3.10.2 Cipher suites	119
3.10.3 Alert protocol	121
3.10.4 Handshake protocol	122
Chapter 4. Crypto hardware and ICSF	133
4.1 Cryptographic hardware	134
4.1.1 CP Assist for Cryptographic Functions (CPACF)	135
4.1.2 Crypto Express 2 feature	135
4.1.3 Comparison of CPACF, CEX2C, and CEX2A	137
4.1.4 Other cryptographic hardware	137
4.2 ICSF	138
4.2.1 ICSF callable services	139

4.2.2 ICSF administration	140
4.3 ICSF services used by CICS WS-Security support	141
4.4 ICSF services used by System SSL	146
Chapter 5. Security scenarios environment	151
5.1 Scenarios: A high level overview	152
5.2 Preparation	159
5.2.1 Software checklist	159
5.2.2 Definition checklist	159
5.3 Basic security configuration	161
5.3.1 Setting up basic security configuration	161
5.3.2 SIT parameters	161
5.4 Creating CA certificates	162
5.4.1 Creating the CICS CA certificate	163
5.4.2 Creating the WebSphere CA Certificate	163
5.4.3 Building the key ring	164
5.4.4 Defining the CICS resources in RACF	165
5.4.5 Modifying the wsbind file using CICS SupportPac CS04	166
5.4.6 Creating the CICS resource definitions	169
5.5 The CICS catalog manager example application	174
5.5.1 Installing and setting up the application	174
5.5.2 Running the application	175
Chapter 6. Enabling SSL	181
6.1 Preparation	182
6.1.1 Software checklist	182
6.1.2 Definition checklist	182
6.1.3 Scenario overview	183
6.2 Creating the CICS server certificate	187
6.3 Configuring WebSphere Application Server for SSL processing	189
6.3.1 Adding the CICS CA certificate to the trust store	189
6.3.2 Configuring WebSphere Application Server's SSL settings	192
6.4 Configuring CICS for SSL processing	196
6.4.1 Enabling CICS to use SSL	197
6.4.2 Defining a new TCPIService	201
6.4.3 Defining a new WEBSERVICE	205
6.4.4 Defining a new URIMAP	206
6.5 Testing the SSL scenario	207
6.6 Creating the WebSphere client certificate	209
6.6.1 Creating the WebSphere certificate	209
6.6.2 Exporting the WebSphere certificate to a file	212
6.6.3 Configuring WebSphere to send the client certificate in SSL	213
6.7 Configuring the CICS service provider for SSL client authentication	214

6.7.1	Configuring the TCPIP SERVICE for SSL client authentication . . .	214
6.7.2	Adding the client certificate to RACF	216
6.7.3	Authorizing the service requester	218
6.8	Testing the SSL client authentication scenario	218
Chapter 7. Signing the SOAP message		221
7.1	Scenario overview	222
7.2	Preparation	224
7.2.1	Software checklist	224
7.2.2	Definition checklist	225
7.3	Setting up certificates and key pairs	226
7.3.1	Validating the cryptography hardware environment	228
7.3.2	Creating the CICS certificate	229
7.3.3	Creating the WebSphere certificate	232
7.3.4	Importing the WebSphere certificate to the key store	232
7.3.5	Adding the CICS CA certificate to the trust store	235
7.4	Configuring the service requester	237
7.5	Configuring CICS for signature processing	243
7.5.1	Enabling CICS to use WS-Security support	243
7.5.2	Defining a new pipeline	244
7.5.3	Authorizing the service requester	247
7.6	Testing the signature scenario	247
7.6.1	SOAP fault messages	253
7.6.2	WebSphere Application Server TCP/IP Monitor	255
Chapter 8. Identity assertion with WebSphere for z/OS		259
8.1	Scenario overview	260
8.1.1	Blind trust model	260
8.1.2	Single sign-on	262
8.2	Preparation	262
8.2.1	Software checklist	262
8.2.2	Definition checklist	263
8.3	Configuring the service requester	264
8.4	Configuring CICS for identity assertion	272
8.4.1	Enabling CICS to use WS-Security support	273
8.4.2	Defining a new pipeline	273
8.4.3	Authorizing the service requester	275
8.5	Testing the identity assertion scenario	275
Chapter 9. Identity assertion with WebSphere DataPower		281
9.1	Scenario overview	282
9.2	Preparation	283
9.2.1	Definition checklist	284
9.2.2	User IDs	285

9.2.3 Certificates	285
9.3 Configuration	286
9.3.1 Configuring the WebSphere service requester	286
9.3.2 Configuring DataPower	286
9.3.3 Configuring CICS	303
9.3.4 Authorizing the service requester	305
9.4 Testing the Identity assertion with WebSphere DataPower scenario ...	306
9.4.1 Running the Web Application	306
9.4.2 Using the Probe facility of DataPower	308
Chapter 10. Enabling WS-Trust with TFIM	313
10.1 Scenario overview	314
10.2 Preparation for this scenario	316
10.2.1 Software checklist	316
10.2.2 Definition checklist	317
10.3 Configuring the WebSphere service requester	318
10.4 Configuring CICS for WS-Trust processing	321
10.4.1 Enabling CICS to use WS-Trust support	321
10.4.2 Defining a new pipeline	322
10.4.3 Authorizing the service requester	325
10.5 Configuring TFIM for the WS-Trust scenario	325
10.5.1 Removing the security constraint from the Trust Service	326
10.5.2 Creating a Trust Chain to process STS requests from CICS ...	328
10.6 Testing the WS-Trust scenario	341
10.7 Using different token types	350
10.7.1 Configuring the service requester	352
10.7.2 Configuring CICS to support different token types	353
10.7.3 Configuring TFIM to validate an LTPA token	353
10.7.4 Testing the WS-Trust with LTPA Token scenario	357
Appendix A. XSLT example	363
Appendix B. Problem determination	369
Appendix C. Sample message handler	379
Related publications	385
IBM Redbooks publications and Redpaper publications	385
Other publications	386
Online resources	386
How to get Redbooks	387
Help from IBM	387
Index	389

Archived

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.


COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

CICS®	RACF®	Tivoli®
CICSplex®	Rational®	WebSphere®
DataPower®	RDN™	z/OS®
DataPower device®	Redbooks®	z10™
DB2®	Redbooks (logo)  ®	z10 EC™
eServer™	S/390®	z9®
IBM®	SupportPac™	zSeries®
Language Environment®	System z9®	
MVS™	System z®	

The following terms are trademarks of other companies:

EJB, J2EE, Java, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Expression, Internet Explorer, Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Preface

Securing access to information is important to any business, especially for business-critical systems that manage sensitive data, as is often the case for systems based on IBM® Customer Information Control System (CICS®). Security becomes even more critical for implementations structured according to service-oriented architecture (SOA) principles, due to loose coupling of services and applications, and their possible operations across trust boundaries.

In this IBM Redbooks® publication, we consider the different ways that CICS Web services can be secured. We consider transport-level security mechanisms such as SSL/TLS and CICS support for the message-based security specifications WS-Security and WS-Trust.

To assist solution and security architects, we outline the main planning considerations and make recommendations on the choice of a security solution. For the systems programmer, we provide detailed setup guidance for configuring common scenarios, including:

- ▶ SSL client authentication
- ▶ Using XML digital signatures
- ▶ Identity assertion
- ▶ Interoperability with WebSphere® DataPower®
- ▶ Using Tivoli® Federated Identity Manager (TFIM) as a Security Token Service

For these scenarios, we provide step-by-step configuration information for CICS and the other involved systems, including WebSphere Application Server, WebSphere DataPower and TFIM.

This book is a companion to the Redbooks publication, *Implementing CICS Web Services*, SG24-7657, which covers other implementation aspects of a CICS Web services solution.

The team that wrote this book

This book was produced by a team of specialists from around the world working at the PSSC Montpellier.

Nigel Williams was the original Project Leader for this book. He is a Certified IT Specialist working at the IBM Design Center in Montpellier, France. He specializes in core business transformation, connectors, and service-oriented architectures. He is the author of several papers and IBM Redbooks® publications, and he speaks frequently on CICS and WebSphere topics. Previously, Nigel worked at the Hursley software lab as a software developer, in systems test, and as customer support for the CICS Early Support Program. He holds a degree in Mathematics and Economics from Surrey University.

Carsten Andersen is a System Designer at Jyske Bank in Denmark. He has more than 25 years of experience in development of IT systems. He holds a degree in Financials from The Danish Bank Academy. His areas of expertise include mainframe development, project management, DB2®, CICS, COBOL and IT infrastructure.

Arnaud Desprets is an IT Architect working for IBM in France. He has 9 years of experience in architecting security solutions for IBM's customers. He holds a degree in Networking from ISEN in France. His area of expertise is SOA security and more specifically in Web Services based technologies and products such as the WebSphere Services Registry and Repository. He also has a great interest in methodology and SOA Governance. He has written extensively on Identity and Access Management.

Tommy Joergensen is a Senior IT Specialist working for IBM Global Services in IBM Denmark. He has more than 25 years of experience working in CICS technical support, including 3 years at IBM Hursley. In recent years he has delivered services at large accounts in Denmark for both the CICS and WebSphere products. Tommy is the IBM representative in the CICS working group of the Nordic Share Guide organization.

James O'Grady is a Systems Tester working for CICS Development in Hursley. He has 8 years of experience in CICS. He has worked at IBM for 2 years. His areas of expertise include CICS Web Services, WS-Security, and CICSplex® SM Workload Management.

Thanks to the following people for their contributions to this project:

Phil Hanson and Mark Cocker of IBM Hursley for supporting this project

Alain Roger, Patrick Sola, and Rene Pons of IBM Montpellier for their technical support

Chris Rayns of the International Technical Support Organization, Poughkeepsie Center for overseeing this book through the production process

Richard M Conway of the International Technical Support Organization, Poughkeepsie Center

Michael Tebolt of IBM Poughkeepsie for assistance with the Tivoli Federated Identity Manager (TFIM) test scenario

Ivan Hargreaves, Mark Pocock, Fraser Bohm, Ian Mitchell and Peter Havercan of IBM Hursley for their review comments

Alain Roessle of IBM Montpellier for assistance with the setup and configuration of WebSphere Application Server for z/OS®, and for his review comments.

Patrick Kappeler of IBM Montpellier and Lennie Dymoke-Bradshaw of IBM UK for their comments on hardware cryptography

Ella Buslovich of the ITSO Poughkeepsie Center for help with the graphics, and Yvonne Lyon of the ITSO San Jose Center for editing this book

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:

ibm.com/redbooks

- ▶ Send your comments in an e-mail to:

redbooks@us.ibm.com

- ▶ Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400



Security for CICS Web services

While you can exercise very strict control over the access to your applications from conventional 3270 terminals, your Web services clients are likely to be in remote locations, and you will be faced with new security issues.

When implementing a CICS Web services solution, you have to consider questions like these:

- ▶ Will *authentication* be done by CICS itself, or in an external server such as WebSphere Application Server?
- ▶ Do you require a trusted third party for issuing and exchanging security tokens?
- ▶ What *authorization* mechanisms will be used to protect access to the CICS system and access to resources, such as transactions, files, and databases?
- ▶ How will you protect the *confidentiality* of data that is transported between the different tiers of the physical configuration?
- ▶ Should you use transport security, for example, SSL/TLS or SOAP message security, to protect your CICS Web services?

In this chapter, we explain what security mechanisms can be used to protect CICS Web services.

1.1 Traditional CICS security

In a CICS environment, the assets you normally want to protect are the application programs and the resources that are accessed by the application programs. To prevent disclosure, destruction, or corruption of these assets, you must control access to the CICS region and to different CICS components.

You can limit the activities of a CICS user to only those functions that the user is authorized to use by implementing one or more of the following CICS security mechanisms:

▶ **Transaction security**

This ensures that users who attempt to run a transaction are entitled to do so.

▶ **Resource security**

This ensures that users who use CICS resources, such as files and transient data queues, are entitled to do so.

▶ **Command security**

This ensures that users who use CICS system programming commands are entitled to do so.

▶ **Surrogate security**

This ensures that a *surrogate* user is authorized to act on behalf of another user.

When CICS security is active, requests to attach transactions, and requests by transactions to access resources, are associated with a user ID. When a user makes such a request, CICS calls the external security manager (such as RACF®) to determine if the user ID has the authority to complete the request. If the user ID does not have the correct authority, CICS denies the request.

In many cases, a user is a human operator, interacting with CICS through a terminal or a workstation. However, the user can also be a Web browser user or, in a Web services solution, a program executing in a client system.

1.1.1 CICS user IDs

When a human operator signs on to a CICS region at the start of a terminal session, he or she is challenged to provide a user ID and password. The user ID remains associated with the terminal until the terminal operator signs off. Transactions executed from the terminal, and requests made by those transactions, are associated with that user ID.

For connections from Web users, there are other ways that the user of a CICS transaction can be identified, including these:

- ▶ An HTTP client can provide HTTP basic authentication information (a user ID and password). The transaction that services the client's request, and further requests made by that transaction, are associated with that user ID.
- ▶ A client program that is communicating with CICS using the Secure Sockets Layer (SSL) supplies a client certificate to identify itself. The security manager maps the certificate to a user ID. The transaction that services the client's request, and further requests made by that transaction, are associated with that user ID.

In addition to these transport-level authentication mechanisms, Web service clients can also pass authentication data, in the form of a security token, within the SOAP message itself. CICS provides direct support for username tokens and X.509 certificates, and can also interoperate with a Security Token Service (STS), such as Tivoli Federated Identity Manager, to provide more advanced authentication of Web services.

1.1.2 Special CICS user IDs

There are two particular user IDs that CICS uses in addition to those that identify individual end users. These are:

Region user ID The CICS region user ID is used for authorization checking when the CICS system (rather than an individual user of the system) requests access to system resources such as CICS data sets and other servers.

Default user ID When a user does not sign on, CICS assigns a default user ID to the user. It is specified in the SIT parameter DFLTUSER. In the absence of more explicit identification, it is used to identify TCP/IP clients that connect to CICS. You should give very little authority to the default user ID.

For a complete discussion of traditional CICS security, refer to *CICS TS V3.2 RACF Security Guide*, SC34-6835.

1.2 Security exposures

An end-to-end security solution addresses the security exposures found along the path of a request from an end client to a target service, including any intermediary services that route, or participate in, the service request.

To illustrate potential security exposures in a Web services environment, we use the bank teller scenario shown in Figure 1-1. The bank teller (Web service requester or client) connects over the Internet to the bank's data center where the Web service provider runs.

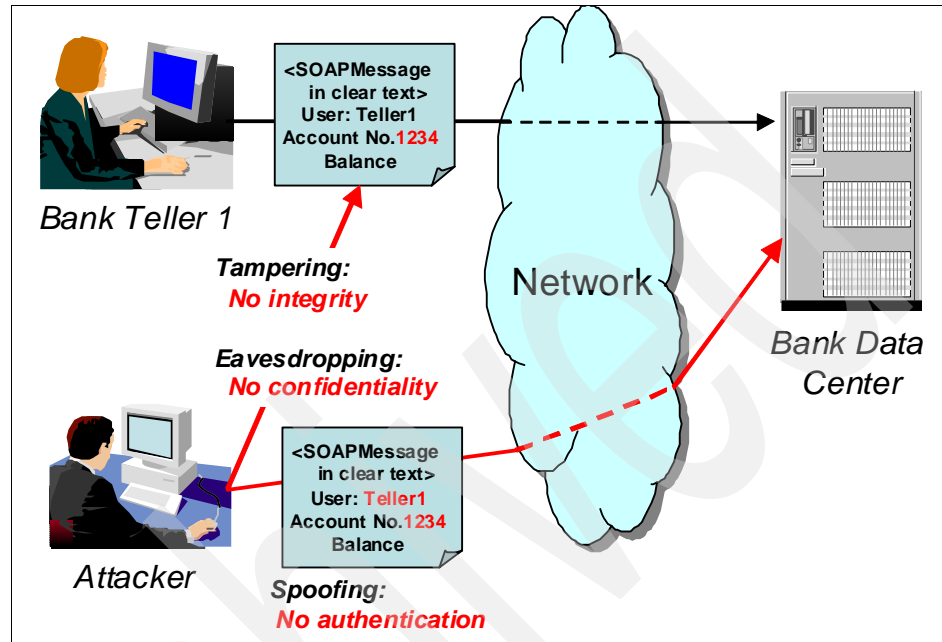


Figure 1-1 Potential security exposures in a Web services environment

If the bank has not applied any security, it has the following exposures:

► **Spoofing**

An attacker posing as the bank teller could send a SOAP message to the service provider to get confidential information or to withdraw money from another customer's account.

The bank can eliminate this security exposure by requiring that the bank teller authenticate herself.

► **Tampering**

An attacker could intercept the SOAP message between the Web service requester and provider and modify the message, for example, to deposit the money into another account by changing the account number. Because there is no integrity constraint, the Web service provider does not verify whether the message has been altered and accepts the modified transaction.

The bank can eliminate this security exposure by implementing digital signatures.

► **Eavesdropping**

An attacker could intercept the SOAP message and read the information contained in the message because it has been sent in clear text. The attacker could obtain confidential customer or bank information such as account numbers and balances.

The bank can eliminate this security exposure by encrypting the SOAP message.

1.3 Security objectives

A complete security solution will put mechanisms in place to achieve the following objectives:

► **Identification**

Identification is the ability to assign an identity to the entity accessing the system. Typically the identity is used to control access to resources. Depending on the security model in which the identification is performed, the identity can be called a *user ID*, a *UID*, or a *principal*.

► **Authentication**

Authentication is the process of validating the identity claimed by the accessing entity. Authentication is performed by verifying authentication information provided with the claimed identity. The authentication information is generally referred to as the accessor's *credentials*. A credential can be the accessor's name and password; it can also be a *token* provided by a trusted party, such as a Kerberos ticket or an X.509 certificate.

You need authentication when you want to give different rights to access resources (such as files and databases) to different requesting identities.

Note: Authentication is usually one of the earliest steps in a request workflow. When authenticated, an identity can be *asserted* to the downstream process steps, meaning that these steps trust the upstream steps to have already successfully authenticated the identity.

► **Authorization**

Authorization is the process of checking whether an identity that has already been authenticated should be given access to a resource that it is requesting. A typical implementation of authorization is to pass to the access control mechanism a *security context* that contains the identity that has been authenticated.

▶ **Integrity**

Integrity ensures that transmitted or stored information has not been altered in an unauthorized or accidental manner. Typically it is a mechanism to verify that what is received over a network is the same as what was sent.

▶ **Confidentiality**

Confidentiality ensures that an unauthorized party cannot obtain the meaning of the transferred or stored data. Typically confidentiality is achieved by encrypting the data.

▶ **Auditing**

With auditing, you capture and record security-related events (such as a user signing onto or off of a system) so that you can analyze them later, perhaps after a breach of your security has occurred.

▶ **Nonrepudiation**

Nonrepudiation means that a sender and a receiver of data are able to provide legal proof to a third party that the sender did send the information, and the receiver received the identical information. Neither side is *able to deny*.

Different IT projects will have different security objectives. Once the specific objectives are understood, you can consider two types of security mechanisms in a Web services environment:

▶ **Transport-level security**

Transport-level security mechanisms such as SSL/TLS can be used to secure Web services. In 1.5, “Transport security” on page 16 we review how different transport-level security mechanisms can be used to secure a CICS Web services solution.

▶ **SOAP message security**

The Web Services Security model introduces a set of interrelated specifications to form a layering approach to security. When products implement these specifications, they send security information within the SOAP message itself as SOAP message headers.

In Chapter 2, “SOAP message security” on page 31 we discuss how SOAP message security can be used to secure a CICS Web services solution.

Before we consider these different types of security, however, we first look at the main CICS resource definitions that help us to secure CICS Web services.

1.4 CICS resources used for securing Web services

In this section, we look at the CICS resources that a systems programmer can define and configure in order to enable security for CICS Web services accessed over HTTP. For information on using WebSphere MQ as a transport, see “WebSphere MQ transport” on page 25.

We do not give a comprehensive definition of these resources, but we focus on the security uses of the resources. Refer to the Redbooks publication, *Implementing CICS Web Services*, SG24-7206 for a complete description of these resources.

1.4.1 CICS as a service provider

Figure 1-2 shows the processing that occurs when a service requester sends a SOAP message over HTTP to a service provider application running in a CICS TS V3.2 region.

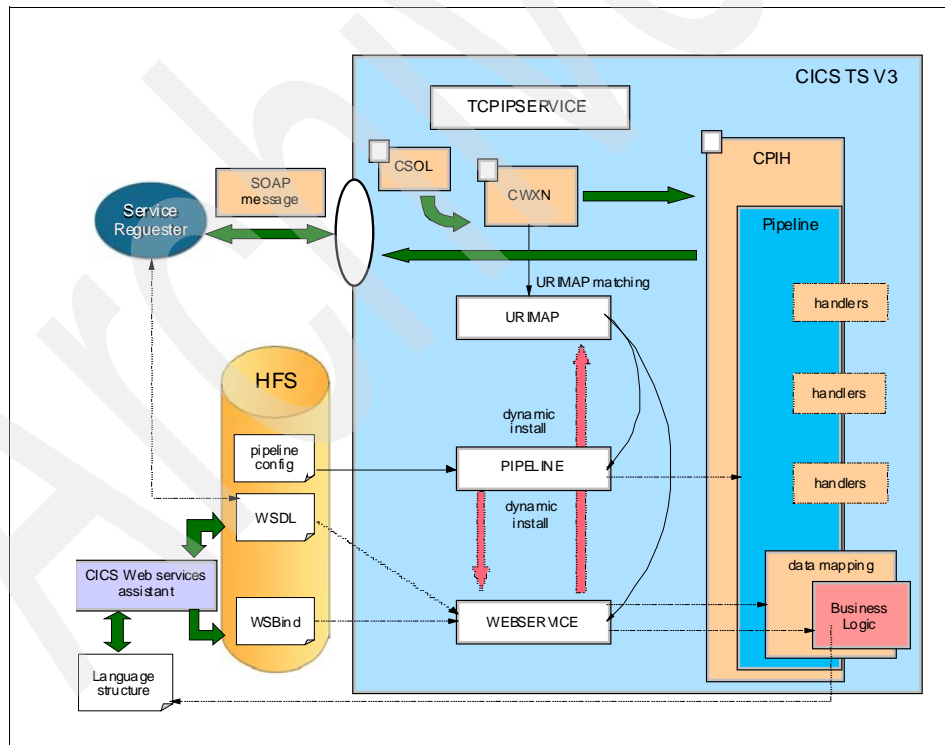


Figure 1-2 Web service run-time service provider processing

The CICS-supplied sockets listener transaction (CSOL) monitors the port specified in the **TCPIPSERVICE** resource definition for incoming HTTP requests. When the SOAP message arrives, CSOL attaches the transaction specified in the TRANSACTION attribute of the TCPIPSERVICE definition; normally, this will be the CICS-supplied Web attach transaction CWXN.

CWXN finds the URI in the HTTP request and then scans the **URIMAP** resource definitions for a URIMAP that has its USAGE attribute set to PIPELINE and its PATH attribute set to the URI found in the HTTP request. If CWXN finds such a URIMAP, it uses the **PIPELINE** and **WEBSERVICE** attributes of the URIMAP definition to get the name of the PIPELINE and WEBSERVICE definitions, which it will use to process the incoming request. CWXN also uses the TRANSACTION attribute of the URIMAP definition to determine the name of the transaction that it should attach to process the pipeline; normally, this will be the CPIH transaction.

CPIH starts the pipeline processing. It uses the PIPELINE definition to find the name of the pipeline configuration file. CPIH uses the pipeline configuration file to determine which message handler programs to invoke.

A message handler in the pipeline (typically, a CICS-supplied SOAP message handler) removes the SOAP envelope from the inbound request and passes the SOAP body to the data mapper function. A security message handler can also be added to the pipeline. A security handler is used to change the security “context” of a request, for example, to change the user ID under which the target request runs.

Note: A *security handler* can be added to a service provider pipeline.

CICS uses channels and containers to pass data between the message handlers of a pipeline. The DFHWS-WEBSERVICE container is used to pass the name of the required WEBSERVICE definition to the data mapper. The data mapper uses the WEBSERVICE definition to locate the main storage control blocks that it needs to map the inbound service request (XML) to a COMMAREA or a container.

The data mapper links to the target service provider application program, providing it with input in the format that it expects. The application program is not aware that it is being executed as a Web service. The program performs its normal processing and then returns an output COMMAREA or container to the data mapper.

The output data from the CICS application program cannot just be sent back to the pipeline code. The data mapper must first convert the output from the COMMAREA or container format into a SOAP body.

Next we provide an overview of how the definition of the main resources (TCPIPSERVICE, URIMAP, PIPELINE, and WEBSERVICE) determine the security mechanisms that are used to secure a CICS service provider application. We then provide practical examples of how to define these resources in the security scenario chapters.

TCPIPSERVICE

A TCPIPSERVICE definition is required in a service provider that uses HTTP or HTTPS as transport. It contains information about the port on which inbound requests are received, and whether any transport based security mechanisms will be applied by CICS.

Note: A *TCPIPSERVICE* definition is used to configure transport security.

Table 1-1 shows the attributes of the TCPIPSERVICE resource definition that impact on the security context within which a service provider application runs.

Table 1-1 Security attributes in TCPIPSERVICE resource

Attribute	Description
AUTHENTICATE	Determines if an authentication and identification scheme is to be used at the transport layer, for example, HTTP basic authentication.
CERTIFICATE	Specifies the label of an X.509 certificate that is used as a server certificate during an SSL handshake.
CIPHERS	Specifies the list of ciphers that this CICS region supports for SSL encryption.
PORTNUMBER	Specifies the number of the port on which CICS is to listen for incoming HTTP or HTTPS requests.
SSL	Specifies whether SSL is used for encryption and authentication.

Further details on these TCPIPSERVICE attributes are provided in “Defining a TCPIPSERVICE resource for SSL” on page 21.

URIMAP

A URI mapping or URIMAP resource definition matches the URIs of Web service requests. URIMAP definitions for inbound Web service requests have a USAGE attribute of PIPELINE. The URIMAP associates a URI for the request with a PIPELINE and WEBSERVICE resource that specifies the processing to be performed.

Importantly, you can use a URIMAP to specify:

- ▶ The name of the transaction that CICS uses for running the pipeline alias transaction (the default is CPIH)
- ▶ The user ID under which the pipeline alias transaction runs

Table 1-2 shows the attributes of the URIMAP resource definition that impact on the security context within which a service provider application runs.

Table 1-2 Security attributes in URIMAP resource for CICS service provider

Attribute	Description
HOST	Specifies the host component of the URI to which the URIMAP definition applies. An example of a host name is www.example.com. This attribute can be used to restrict Web service requests to specific host names.
PIPELINE	Specifies the name of the PIPELINE resource definition for the Web service. The PIPELINE resource definition provides information about the message handlers, including security messages handlers, which act on the service request from the client.
SCHEME	Specifies the scheme component of the URI to which the URIMAP definition applies, which is either HTTP (without SSL) or HTTPS (with SSL). It can be used to restrict Web service requests to HTTPS only.
TCPIPSERVICE	Specifies the name of a TCPIPSERVICE resource definition, that defines an inbound port to which this URIMAP definition relates. It can be used to restrict access to Web services through a specific TCPIPSERVICE and its associated transport based security mechanisms.
TRANSACTION	Specifies the name of the pipeline alias transaction that is to be used to start the pipeline. This is a very important attribute as it directly controls the transaction identifiers that are used for Web service requests and that therefore need to be protected using transaction security.
USAGE	Must specify PIPELINE to indicate that this URIMAP definition applies to inbound Web service requests.
user ID	Specifies the user ID under which the pipeline alias transaction is attached. Important: A user ID that you specify in the URIMAP definition is overridden by any user ID that is obtained directly from the client.
WEBSERVICE	Specifies the name of the Web service.

When you install a PIPELINE resource, CICS scans the directory specified in the PIPELINE's WSDIR attribute (the pickup directory) for wsbind files, and creates URIMAP and WEBSERVICE resources dynamically. If you want to use the URIMAP definition to specify either the name of the transaction or the user ID under which the pipeline will run, you can either:

- ▶ Specify the TRANSACTION and user ID parameters when using the CICS Web services assistant to create the Web service.
- ▶ Use CICS SupportPac™ CS04 *CICS TS for z/OS: WSBind File Display and Change Utility* to change an existing wsbind file.
- ▶ Create and install your own URIMAP resource definition.

Examples of these techniques are provided in the security scenarios section of this book.

PIPELINE

A PIPELINE resource definition provides information about the message handlers that act on a service request and on the response. The information about the message handlers is supplied indirectly; the CONFIGFILE attribute of the PIPELINE definition specifies the name of an HFS file, called the pipeline configuration file, which contains an XML description of the message handlers and their configuration.

Pipeline configuration file

There are two kinds of pipeline configuration files: one describes the configuration of a service provider pipeline, the other describes the configuration of a service requester pipeline. Each is defined by its own schema, and each has a different root element. The root element for a provider pipeline is <provider_pipeline>, while the root element for a requester pipeline is <requester_pipeline>.

Example 1-1 shows the sample pipeline configuration file `basicsoap12provider.xml` provided by CICS.

Example 1-1 CICS-supplied sample pipeline configuration file basicsoap12provider.xml

```
<?xml version="1.0" encoding="EBCDIC-CP-US"?>
<provider_pipeline
  xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline provider.xsd">
  <service>
    <terminal_handler>
      <cics_soap_1.2_handler/>
    </terminal_handler>
```

```
</service>  
<apphandler>DFHPITP</apphandler>  
</provider_pipeline>
```

To configure a pipeline to support SOAP message security, you must add a security handler to your pipeline configuration file. You can use the CICS-supplied security handler or create your own. For more information on using the CICS-supplied security handler, see “CICS and SOAP message security” on page 35. For more information on writing your own security handler see “Custom security handlers” on page 53.

A security handler is most often used to change the security “context” of a request, based on the contents of the SOAP message. The CICS-supplied security handler can also be used for XML digital signature processing and XML encryption.

If a security handler changes the contents of the DFHWS-USERID container, the target application will execute in a new task that is associated with the new user ID. If a security handler changes the contents of container DFHWS-TRANID, the target application will execute in a new task that is associated with the new transaction ID.

WEBSERVICE

A WEBSERVICE resource defines the aspects of the run time environment for a CICS application program deployed as a Web service. Three objects define the execution environment that allows a CICS application program to operate as a Web service provider:

- ▶ The Web service description
- ▶ The Web service binding file
- ▶ The pipeline

These three objects are defined to CICS on the following attributes of the WEBSERVICE resource definition:

- ▶ WSDLFILE
- ▶ WSBIND
- ▶ PIPELINE

The WEBSERVICE definition does not have a direct impact on the security context used for processing the request.

Setting the user ID

It is possible that for a single Web service request transported by HTTP, multiple methods for setting the user ID will be used at the same time. In this event, CICS uses the following order of precedence for determining the user ID under which the target business logic program runs:

1. A user ID specified by a message handler, or a SOAP header processing program, that is included in the pipeline that processes the SOAP message
2. A user ID obtained from the Web client using basic authentication, or a user ID associated with a client certificate
3. A user ID specified in the URIMAP definition for the request
4. The CICS default user ID, if no other user ID can be determined

Important: When several identification mechanisms are used for the same request, SOAP message security takes precedence.

It might be meaningful to run a Web service request under two user IDs, for example:

- ▶ An initial user ID, which is set on the URIMAP definition or obtained using a transport-based authentication process.
If a transport based authentication mechanism, such as SSL client authentication is used, this identity can be considered to be the identity of the server that initiated the request.
- ▶ A second user ID, which is set from the security token contained in the SOAP message. This identity is often the caller's (or service requester's) identity.

This dual identity process allows general authorizations to be given to connected servers and more specific service authorizations to be given to specific groups of service requesters. This requires two tasks for processing a single Web service request because a single CICS task can only be associated to a single user ID.

1.4.2 CICS as a service requester

Figure 1-3 shows the processing that occurs when a service requester running in a CICS TS V3.2 region sends a SOAP message over HTTP to a service provider.

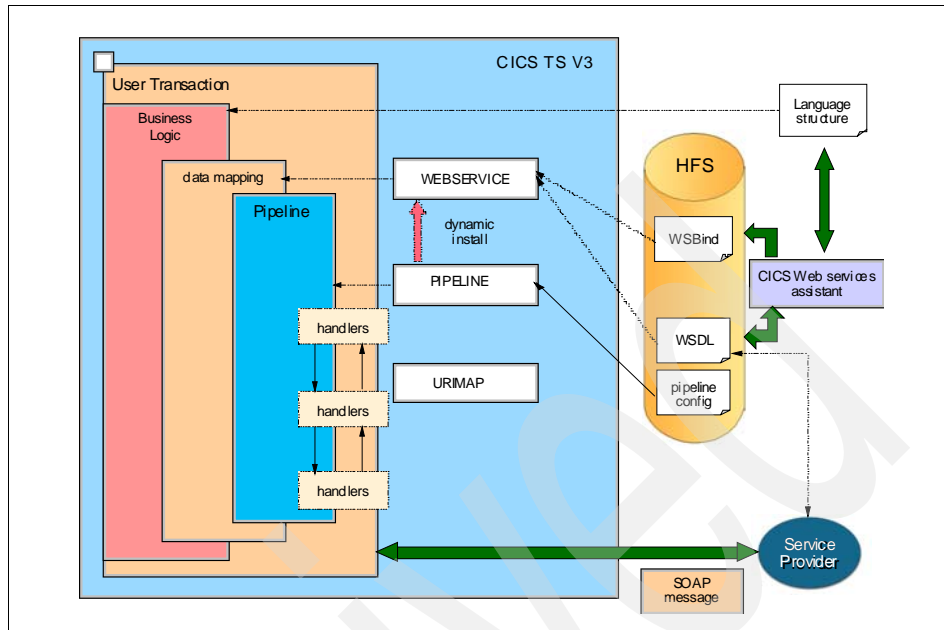


Figure 1-3 Web service run-time service requester processing

When the service requester issues the EXEC CICS INVOKE WEBSERVICE command, CICS uses the information found in the wsbind file that is associated with the specified WEBSERVICE definition to convert the language structure into an XML document. CICS then invokes the message handlers specified in the pipeline configuration file, and they convert the XML document into a SOAP message. The CICS-supplied security handler can be added to a service requester pipeline, for example, in order to add a security token to the outbound SOAP message. You can also add your own security handlers.

Note: A *security handler* can be added to a service requester pipeline.

CICS sends the SOAP request message to the remote service provider via either HTTP or HTTPS. A URI can be specified on the INVOKE WEBSERVICE command. If specified, it supersedes any URI specified in the WEBSERVICE resource definition. A URIMAP resource can be used to specify transport related security information (see “URIMAP” on page 15).

When the SOAP response message is received, CICS passes it back through the pipeline. The message handlers extract the SOAP body from the SOAP envelope, and the data mapping function converts the XML in the SOAP body into a language structure, which is passed to the application program in container DFHWS-DATA.

Next we provide an overview of how the definition of the URIMAP and PIPELINE resources allows us to determine the security mechanisms that can be used to secure the request from a CICS service requester application.

URIMAP

URIMAP definitions for requests from CICS as an HTTP client have a USAGE attribute of CLIENT. These URIMAP definitions specify URLs that are used when a user application, acting as a Web service requester, makes a request through CICS Web support to a remote service provider.

You can use a URIMAP to specify SSL related security information.

Table 1-3 shows the attributes of the URIMAP resource definition that impact on the security context within which a service provider application runs.

Table 1-3 Security attributes in URIMAP resource for CICS service requester

Attribute	Description
CERTIFICATE	Specifies the label of the X.509 certificate that is to be used as the SSL client certificate during the SSL handshake. It is up to the server to request an SSL client certificate, and if this happens, CICS supplies the certificate identified by the label that is specified in the CERTIFICATE attribute. If this attribute is omitted, the default certificate defined in the key ring for the CICS region user ID is used. The certificate must be stored in the key ring used by the CICS region.
CIPHERS	Specifies the list of ciphers that this CICS region supports for SSL encryption.
HOST	Specifies the host component of the URI to which the URIMAP definition applies.
SCHEME	Specifies the scheme component of the URI to which the URIMAP definition applies, which is either HTTP (without SSL) or HTTPS (with SSL).
USAGE	Specify CLIENT to indicate that this URIMAP definition applies to outbound Web service requests.

PIPELINE

A service requester pipeline configuration file can include security message handlers. You can use the CICS-supplied security handler DFHWSSE1 for signing the outbound message, for XML encryption or for attaching a security token to the message. You can also use your own self-written security handler.

1.5 Transport security

In this section we review how basic authentication and SSL/TLS can be used to secure a CICS Web services solution when HTTP is the transport and also when WebSphere MQ is the transport.

1.5.1 HTTP transport

When a CICS Web service is invoked using HTTP, you can use basic authentication to authenticate the Web service client, and you can use SSL/TLS to both authenticate the Web service client and to ensure message integrity and confidentiality.

Basic authentication

HTTP basic authentication is a simple challenge and response mechanism with which a server can request authentication information (a user ID and password) from a client. The client passes the authentication information to the server in an HTTP Authorization header. The authentication information is in base-64 encoding.

The Basic Authentication protocol is specified in RFC2617, at:

<http://www.ietf.org/rfc/rfc2617.txt>

The AUTHENTICATE attribute on the CICS TCPIP SERVICE resource definition specifies the authentication and identification scheme to be used for inbound TCP/IP connections for the HTTP protocol. You enable HTTP basic authentication by specifying BASIC for the AUTHENTICATE attribute.

A CICS service provider application can be protected by HTTP basic authentication. However, the HTTP basic authentication scheme can only be considered a secure means of authentication when the connection between the Web service client and the CICS region is secure. If the connection is insecure, the scheme does not provide sufficient security to prevent unauthorized users from discovering and using the authentication information for a server. If there is a possibility of a password being intercepted, basic authentication should be used in combination with SSL/TLS, so that SSL encryption is used to protect the user ID and password information.

CICS does not provide base support for a service requester application to use HTTP basic authentication to authenticate with an external service provider application, however, this support is provided in SupportPac CA&J: *Generate Basic Authentication headers for a CICS HTTP client*. This SupportPac provides sample code to generate an HTTP Authorization header from a CICS HTTP

client application. The header is then used to send basic authentication credentials. The SupportPac consists of several sample programs, including a sample Cobol program that can be used as a transport handler in a CICS Web Services requester pipeline.

CICS support for SSL/TLS

The Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols provide confidentiality and data integrity between two applications communicating over the Internet. CICS uses System SSL to support both the SSL 3.0 and TLS 1.0 protocols.

A CICS service provider application can be secured using HTTPS (HTTP over a SSL/TLS connection), and a CICS service requester application can use HTTPS to invoke a service provider application. HTTPS connections will automatically use the TLS 1.0 protocol, unless the client specifically requires SSL 3.0. For a full explanation of how SSL/TLS works, refer to 3.10, “Transport Layer Security (TLS) 1.0 protocol” on page 116.

HTTPS has the following advantages:

- ▶ It provides a fast and secure transport for CICS Web services.
- ▶ It provides for authentication through either HTTP basic authentication or a client X.509 certificate.
- ▶ It provides integrity for the data passed between the service requester and the service provider.
- ▶ It provides confidentiality for the data passed between the service requester and the service provider by using efficient secret key cryptography.
- ▶ It can be used with hardware cryptographic devices that can significantly reduce the cost of SSL handshakes. You can customize your encryption settings to use only the cipher suites that use the Integrated Cryptographic Service Facility (ICSF). See “ICSF” on page 138 for information about ICSF.
- ▶ It is mature and similarly implemented by most vendors, and therefore, is subject to few interoperability problems.

To activate SSL/TLS support in a CICS TS V3.2 region, you must perform the following tasks:

- ▶ Obtain or create a server certificate if CICS is the service provider.

You can obtain an X.509 certificate from a certificate authority such as Verisign, whose Web page is at:

<http://www.verisign.com/>

Alternatively, you can use RACF to create certificates. For more information see the *z/OS Security Server RACF Security Administrator's Guide*, SC34-6835. If you create server certificates using RACF, you must configure your clients to ensure that they will recognize the server certificate.

If CICS is the service requester, you might need to obtain a client certificate.

- ▶ Create a key ring.

In CICS, the required server certificate and related information about certificate authorities are held in a key ring in the RACF database. The key ring contains your system's private and public key pair, together with your server certificate and the certificates for all the certificate authorities that might have signed the certificates you receive from your clients.

The RACDCERT command installs and maintains public key infrastructure (PKI) private keys and certificates in RACF. RACF supports multiple PKI private keys and certificates to be managed as a group. These groups are called *key rings*.

Optionally, you can designate one certificate in the key ring as the default certificate. When a client or server requests a certificate from CICS, the default certificate is used unless you have specified otherwise:

- For inbound HTTPS requests, specify the certificate in the TCPIP SERVICE resource definition.
- For outbound HTTPS requests, specify the certificate in the URIMAP resource definition.

The CICS TS V3.2 RACF Security Guide provides more information on creating key rings and certificates. We also show practical examples in our tested security scenarios (see 5.4, "Creating CA certificates" on page 162).

- ▶ Ensure that your CICS region has access to the z/OS System SSL library.

The CICS region must have access to the z/OS System SSL library SIEALNKE by means of the STEPLIB or JOBLIB statements, or by using the system link library, as appropriate.

- ▶ Specify values for the CICS system initialization parameters related to SSL.

We discuss these parameters in the next section.

- ▶ Define a TCPIP SERVICE resource for inbound requests
- ▶ Optionally define a URIMAP resource for outbound requests.

System initialization parameters related to SSL

To activate SSL/TLS support in a CICS TS V3.2 region, you must specify values for the following system initialization parameters:

▶ CRLPROFILE=*PROFILENAME*

Specifies the 246-character uppercase name of the profile that will be used to authorize CICS to access the certification revocation lists (CRLs) that are stored in an LDAP server. If the CRLPROFILE parameter is omitted or invalid, or the specified profile contains invalid data, or if the LDAP server identified by the profile is unavailable, CICS does not check the revoked status of certificates during SSL handshakes.

▶ ENCRYPTION={STRONG | WEAK | MEDIUM}

Specifies the cipher suites that CICS uses for secure TCP/IP connections. When a secure connection is established between a service requester and a service provider, the most secure cipher suite supported by both is used.

- Use ENCRYPTION=STRONG when you can tolerate the overhead of using high encryption if the other system requires it.
- Use ENCRYPTION=WEAK when you want to use encryption keys up to 40 bits in length.
- Use ENCRYPTION=MEDIUM when you want to use encryption keys up to 56 bits in length.

When you use the CICS CEDA transaction to define a TCPIP SERVICE or URIMAP resource, CICS automatically initializes the CIPHERS attribute of that resource definition with a default list of acceptable cipher suites; the contents of the default list depends on the value of the ENCRYPTION parameter.

▶ KEYRING=*keyring-name*

Specifies the name of a key ring in the RACF database that contains keys and certificates used by CICS. It must be owned by the CICS region user ID.

When CICS finds the KEYRING parameter in the system initialization table, it knows that SSL/TLS processing is required and creates one open transaction environment (OTE) TCB, called the SP TCB, that is used to own socket pthread tasks. The SP TCB manages a pool of S8 TCBs that are used to process SSL connections. Each SSL connection uses an S8 TCB, which is allocated from the SSL pool and requires a UNIX® pthread. All of the S8 TCBs run within a single LE enclave, which is owned by the SP TCB and contains the SSL cache.

▶ MAXSSLTCBS={& | *number*}

Specifies the maximum number of S8 TCBs that are available to CICS to process SSL connections. The S8 TCBs are created and managed in the SSL pool.

S8 TCBs are locked to a transaction only for the amount of time that it needs to perform SSL functions. After the SSL negotiation is complete, the TCB is released back into the SSL pool to be reused.

Increasing the number of available TCBs allows more simultaneous SSL connections to take place. However, increasing the number of TCBs too much will impact storage below the line.

The maximum value that you can specify for the MAXSSLTCBS parameter is 1024.

► **SSLDELAY={600 | *number*}**

Specifies the length of time in seconds for which CICS retains session IDs for secure socket connections in the SSL cache. Session IDs are tokens that represent a secure connection between CICS and an SSL client. The session ID is created and exchanged between the SSL client and CICS during the SSL handshake.

While the session ID is retained by CICS within the SSLDELAY period, CICS will re-establish an SSL connection with a client by using only a partial handshake as discussed in “Resuming a session” on page 131. The value is a number of seconds in the range 0 through 86400. The default value is 600.

Increasing the value of the SSLDELAY parameter retains the session IDs in the cache for longer, thereby optimizing the time it takes to perform SSL negotiations.

The SSLDELAY parameter only applies when the SSLCACHE parameter has the value CICS.

► **SSLCACHE={CICS | SYSPLEX}**

Specifies whether CICS should use the local SSL cache in the CICS region, or share the cache across multiple CICS regions by using the sysplex session cache support provided by System SSL.

When SSLCACHE=CICS, a client who successfully connects to CICS region 1 on z/OS system 1 and then connects to CICS region 2 on z/OS system 2 must go through a full SSL handshake in both cases; this is because CICS stores the SSL session id in a cache that is local to the CICS address space.

When SSLCACHE=SYSPLEX, an SSL session established with a CICS region on one system in the sysplex can be resumed using a CICS region on another system in the sysplex as long as the SSL client presents the session identifier obtained for the first session when initiating the second session.

CICS uses the sysplex session cache support provided by the System SSL started task (GSKSRVR), an optional component of System SSL. GSKSRVR processes the following environment variables:

- GSK_LOCAL_THREADS

This variable specifies the maximum number of threads which will be used to handle program call requests from SSL applications running on the same system as the GSKSRVR started task.

- GSK_SIDCACHE_SIZE

This variable specifies the size of the sysplex session cache in megabytes.

- GSK_SIDCACHE_TIMEOUT

This variable specifies the sysplex session cache entry timeout in minutes.

Sharing SSL session IDs across different CICS regions is particularly useful when Web service requests are being routed across a set of CICS regions using TCP/IP connection workload balancing techniques, such as TCP/IP port sharing or Sysplex Distributor. If the cache is shared between the CICS regions, the number of full SSL handshakes can be significantly reduced.

In order to use the sysplex session cache, each system in the sysplex must be using the same external security manager (for example, z/OS Security Server RACF) and a user ID on one system in the sysplex must represent the same user on all other systems in the sysplex (that is, user ID ZED on System A has the same access rights as user ID ZED on System B).

The external security manager must support the functions, RACROUTE REQUEST=EXTRACT,TYPE=ENVRXTR and RACROUTE REQUEST=FASTAUTH. Refer to *System SSL Programming*, SC24-5901, for additional information about the System SSL started task.

Caching across a sysplex can only take place when the regions accept SSL connections at the same IP address.

In Chapter 6., “Enabling SSL” on page 181 we show how we enabled our CICS region to support SSL connections from our Web service client running in WebSphere Application Server.

Defining a TCPIPSERVICE resource for SSL

The following attributes of the TCPIPSERVICE resource definition relate to using SSL for inbound Web service requests when the transport is HTTP:

- ▶ AUTHENTICATE(NO | BASIC | CERTIFICATE | AUTOREGISTER | AUTOMATIC)

Specifies the authentication and identification scheme to be used.

- NO

The client is not required to send authentication or identification information. However, if the client sends a valid certificate that is already

registered to the security manager, and associated with a user ID, then the user ID identifies the client.

– BASIC

HTTP basic authentication is used to obtain a user ID and password from the client.

– CERTIFICATE

SSL client certificate authentication is used to authenticate and identify the client. The client must send a valid certificate that is already registered to RACF and associated with a user ID. If a valid certificate is not received, or the certificate is not associated with a user ID, the connection is rejected.

When the end user has been successfully authenticated, the user ID associated with the certificate identifies the client.

Note: If you specify AUTHENTICATE(CERTIFICATE) or AUTHENTICATE(AUTOREGISTER), you must specify SSL(CLIENTAUTH).

– AUTOREGISTER

SSL client certificate authentication is used to authenticate and identify the client.

- If the client sends a valid certificate that is *not* registered to the security manager, then HTTP Basic authentication is used to obtain a user ID and password from the client. If the password is valid, CICS registers the certificate with the security manager and associates it with the user ID. The user ID identifies the client.
- If the client sends a valid certificate that *is* already registered to the security manager, and associated with a user ID, then that user ID identifies the client.

– AUTOMATIC

This combines the AUTOREGISTER and BASIC functions.

- If the client sends a certificate that is already registered to the security manager, and associated with a user ID, then that user ID identifies the client.
- If the client sends a certificate that is not registered to the security manager, then HTTP Basic authentication is used to obtain a user ID and password from the client. Provided that the password is valid, CICS registers the certificate with the security manager, and associates it with the user ID. The user ID identifies the client.

- If the client does not send a certificate, then HTTP Basic authentication is used to obtain a user ID and password from the user. When the end user has been successfully authenticated, the user ID supplied identifies the client.

► CERTIFICATE

Specifies the label of an X.509 certificate that is used as a *server* certificate during the SSL handshake. If this attribute is omitted, the default certificate defined in the key ring for the CICS region user ID is used.

► CIPHERS

Specifies a string of up to 56 hexadecimal digits that is interpreted as a list of up to 28 2-digit cipher suite numbers (the tables in “Cipher suites” on page 119 show the cipher suite numbers assigned to each cipher suite).

When you use CEDA to define the resource, CICS automatically initializes this attribute with a default list of acceptable numbers. The contents of the default list depends on the level of encryption that is specified by the ENCRYPTION system initialization parameter.

- For ENCRYPTION=WEAK, the default value is 03060102.
- For ENCRYPTION=MEDIUM, the default value is 0903060102.
- For ENCRYPTION=STRONG, the default value is 050435363738392F303132330A1613100D0915120F0C03060201.

Note: The default cipher list is dependent on the level of z/OS. When you upgrade z/OS you can use CEDA ALTER to clear the cipher list; it will then be updated to the default cipher list supported by the current z/OS level.

We discuss cipher suites in more detail in Chapter 3.1, “The role of cryptography” on page 70.

You can customize the default list of cipher suites to set a minimum level as well as a maximum level of encryption to be used in the encryption negotiation process of the SSL/TLS handshake.

Important: If the SSL handshake negotiates down to using cipher suite 01 or 02, there is no encryption and data will be transmitted in the clear. If you require encryption, you might therefore want to remove 01 and 02 from the list of cipher suites.

► PORTNUMBER(port)

Specifies the number of the port on which CICS is to listen for incoming HTTPS requests.

► SSL(NO | YES | CLIENTAUTH)

Specifies whether the TCPIP SERVICE is to use SSL for encryption and authentication.

- NO
SSL is not to be used
- YES
An SSL session is to be used; CICS will send a server certificate to the client.
- CLIENTAUTH
An SSL session is to be used; CICS will send a server certificate to the client, and the client must send a client certificate to CICS.

Using hardware cryptographic features with System SSL

The use of cryptographic hardware by System SSL is required in order to maximize performance of using SSL/TLS with CICS. During its runtime initialization processing, System SSL checks to see what cryptographic hardware is available. Whenever possible, it will use the hardware rather than its own software algorithms to perform a cryptographic algorithm.

In “Cryptographic hardware” on page 134 we review the different cryptographic hardware options and provide information on how CICS SSL support takes advantage of cryptographic hardware.

Optimizing SSL

Implementing SSL will cause an increase in CPU usage. You should only use SSL for applications that need this level of security. For these applications, you should consider the following techniques for optimizing the performance of SSL in your environment:

- Utilizing cryptographic hardware.
- Increasing the value of the CICS system initialization table parameter SSLDELAY so the session IDs remain in the SSLCACHE longer, which will result in only partial SSL handshakes.
- Increasing the value of the CICS system initialization table parameter MAXSSLTCBS so there are more S8 TCBs in the SSL pool for the SSL handshake negotiation.
- Using the CICS SSLCACHE system initialization table parameter to implement SSL caching across a sysplex if Web service requests are being routed across a set of CICS regions. However, if you are using a single CICS region then you should specify SSLCACHE(CICS) as opposed to SSLCACHE(SYSPLEX) in order to avoid the additional cost of making the SSL session ID shareable.

- ▶ Keeping the socket open by coding SOCKETCLOSE NO on the TCPIPSERVICE definition. This is the default for HTTP 1.1 persistent sessions and removes the need to perform an SSL handshake on the second or subsequent HTTP request.
- ▶ Only using client authentication by specifying SSL(CLIENTAUTH) on the TCPIPSERVICE definition when you really need your clients to identify themselves with a client certificate. Client authentication requires more network transmissions during the SSL handshake, and more processing by CICS to handle the received certificate.

1.5.2 WebSphere MQ transport

Figure 1-4 shows the processing that occurs when a service requester sends a SOAP message over WebSphere MQ to a service provider application running in a CICS TS V3.2 region.

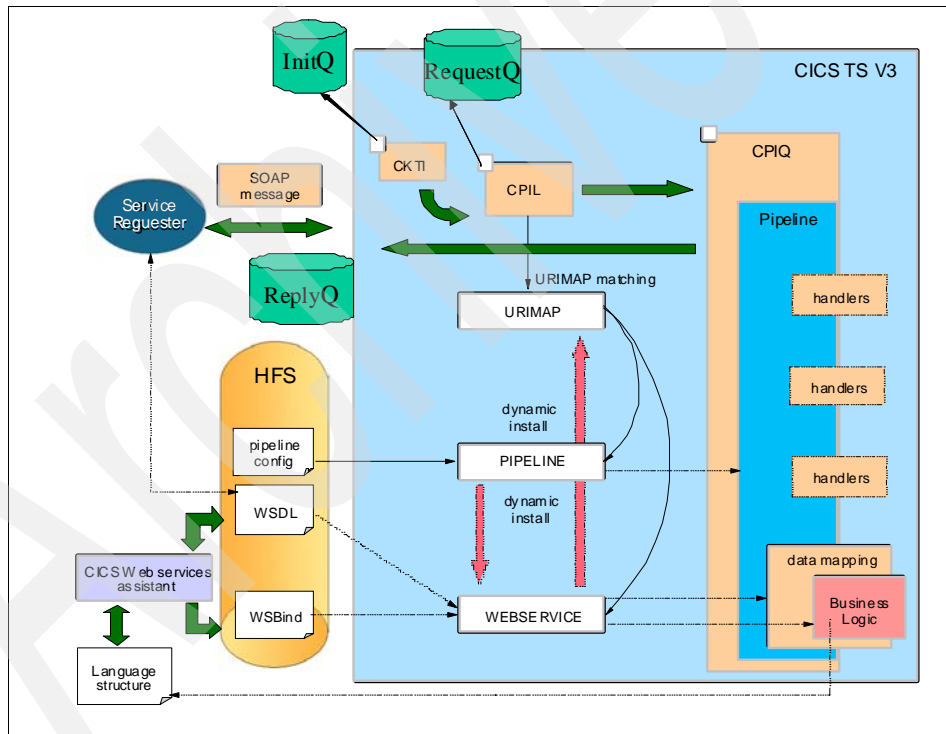


Figure 1-4 Web service run-time service provider processing for WebSphere MQ

The CICS-supplied WebSphere MQ trigger monitor transaction (CKTI) issues an MQGET command (with the wait option) for the initiation queue. This performs an analogous function to the sockets listener transaction, in that it waits for a message to arrive on the queue and then starts further processing. The initiation queue is defined as a local queue to the queue manager that is accessed by the CICS region.

A SOAP message is sent via WebSphere MQ and arrives at a local queue which is set to trigger a WebSphere MQ process for every message that arrives. The process puts a message onto the initiation queue.

The message on the initiation queue is then retrieved by the CKTI task. The message includes information from the process definition including an application id that is used by CICS to set the transaction ID of the SOAP WebSphere MQ inbound listener transaction (the default transaction ID is CPIL).

The CPIL task opens the request queue that contains the SOAP message and retrieves the message. The message contains a URI, specified either as an MQHRF2 header or as trigger data on the initiation queue. CPIL then scans the URIMAP resource definitions for a URIMAP that has a USAGE attribute set to PIPELINE and a PATH attribute set to the URI found in the MQ message.

If CPIL finds a matching URIMAP, it uses the PIPELINE and WEBSERVICE attributes of the URIMAP definition to get the name of the PIPELINE and WEBSERVICE definitions, which it then uses to process the incoming request.

Note: The Host attribute of the URIMAP is used by CICS to match the name of the request queue. However, queue names are case-sensitive whilst URIMAP hostnames are lower-case by default. Specifying host as '*' avoids this problem.

CPIL also uses the TRANSACTION attribute of the URIMAP definition to determine the name of the transaction that it should attach to process the pipeline (the default transaction ID is CPIQ).

Note: If the URIMAP specifies the CPIH transaction, then CPIQ will be used in its place.

CPIQ starts the pipeline processing. It uses the PIPELINE definition to find the name of the pipeline configuration file and to determine which message handler programs to invoke. From this point onwards, the processing is exactly the same as for SOAP over HTTP requests.

Securing access to WebSphere MQ resources

As with CICS, RACF can be used to secure access to WebSphere MQ resources. Whereas CICS uses System Initialization Parameters such as SEC, XTRAN and CMDSEC to determine what checks are active at runtime, WebSphere MQ checks for certain RACF classes and profiles within those classes to determine what security checks to perform.

If security for a CICS region is required then the SIT parameter SEC must be set to YES. This is done on a CICS region by region basis. For WebSphere MQ, the RACF class MQADMIN is used to activate security for WebSphere MQ resources. Since RACF classes are shared across a Sysplex, there needs to be some mechanism for turning off WebSphere MQ security, for example, in a test system.

WebSphere MQ uses RACF profiles called *switch profiles* to toggle security at lower levels than the sysplex. Switch profiles contain the name of an individual queue manager or queue sharing group. If security for the particular queue manager is not required, then WebSphere MQ sets its internal subsystem security switch off and performs no security checks. The sequence of checks to determine whether or not security is active for a particular queue manager or queue-sharing group is shown in Figure 1-5.

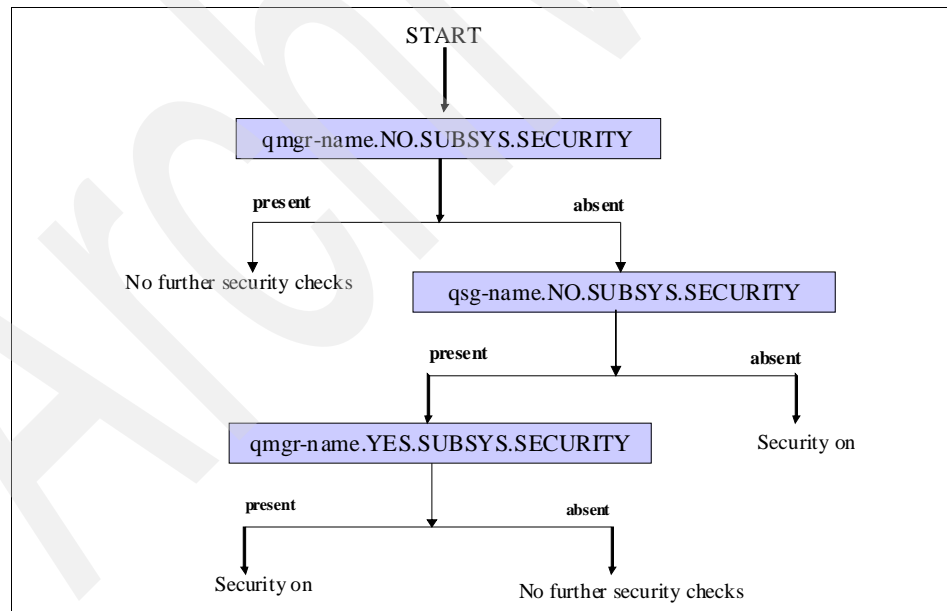


Figure 1-5 Sequence for deciding if security is on for WebSphere MQ

Queue manager level checks are performed first. If the queue manager is not a member of a queue-sharing group, then no queue-sharing group checks are made.

If security checking is activated for a queue manager, WebSphere MQ checks that certain other RACF classes are active. Whereas CICS uses SIT parameter (for example XTRAN and XCMD) to determine which CICS resources to protect, WebSphere MQ checks the existence of specific RACF classes directly. The RACF classes MQCONN, MQCMDS, MQQUEUE, MQPROC and MQNLIST control what resource level checks are performed. For example, MQCONN controls connection security (can I connect to this queue manager?) and MQQUEUE controls queue resource security (which queues can I access and how?).

When using WebSphere MQ as the transport mechanism for accessing CICS Web services, you need to consider the following points:

- ▶ The SOAP WebSphere MQ inbound listener transaction (default CPIL) is started by the trigger monitor using the same user ID as the trigger monitor transaction. This user ID must have UPDATE authority to the request queue and the backout queue (if one is specified).
- ▶ If AUTH=IDENTIFY is specified in the USERDATA parameter of the WebSphere MQ process definition for CPIL, then the user ID under which CPIL runs must have surrogate authority to allow it to start transactions on behalf of the user IDs contained in the WebSphere MQ message descriptors (MQMDs) of the messages.

More information about security for WMQ can be found in *WebSphere MQ Security*, SC34-6588 and in *WebSphere MQ Security in an Enterprise Environment*, SG24-6814.

SSL/TLS with WebSphere MQ

SSL/TLS can be used to secure SOAP messages that are transported using WebSphere MQ. WebSphere MQ supports both the SSL 3.0 and TLS 1.0 protocols. You specify the cryptographic algorithms that are used by the SSL protocol by supplying a CipherSpec as part of the channel definition.

See *WebSphere MQ Security*, SC34-6588 for more information on using SSL/TLS with WebSphere MQ.

Determining the user ID when using WebSphere MQ

It is possible that for a single Web service request transported by WebSphere MQ, multiple methods for setting the user ID will be used at the same time. In this event, CICS uses the following order of precedence to determine the user ID under which the target business logic program runs:

1. A user ID specified by a message handler, or a SOAP header processing program, that is included in the pipeline that processes the SOAP message.

For example, a SOAP header processing program could extract a username from the SOAP message and specify that the CICS task should run with this user ID.
2. A user ID obtained from the MQ message descriptor

A message can contain message context information, such as a user ID. This information is held in the message descriptor and can be generated by the queue manager when a message is put on a queue by an application, or by the application itself. This allows the receiving application to run with the same identity as the application that put the message on the queue.
3. A user ID specified on the URIMAP definition.
4. The CICS default user ID, if no other user ID can be determined

Archived

SOAP message security

CICS provides support for a number of related specifications that enable you to secure SOAP messages, including the WS-Security specification. Support has been enhanced with CICS TS V3.2 to include an implementation of the Web Services Trust Language (or WS-Trust) specification.

CICS can now interoperate with a Security Token Service (STS), such as Tivoli Federated Identity Manager, to validate and issue security tokens in Web services. This enables CICS to send and receive messages that contain a wider variety of security tokens.

In this chapter, we provide an overview of the SOAP message security specifications that CICS supports, and we explain how you configure the CICS-supplied security handler.

We also provide practical guidance that will help you to choose which security technologies you should use to secure your own CICS Web services infrastructure.

2.1 WS-Security

The first version of the WS-Security specification was proposed by IBM, Microsoft®, and VeriSign in April 2002. After the formalization of the April 2002 specification, the specification was transferred to the OASIS consortium:

<http://www.oasis-open.org>

The latest core specification, Web Services Security: SOAP Message Security 1.0 (WS-Security 2004) was standardized in March 2004.

WS-Security provides a foundational set of SOAP message extensions for building secure Web services by defining new elements to be used in the SOAP header for message-level security. It specifies the use of *security tokens*, *digital signatures*, and *XML encryption* to protect and authenticate SOAP messages. It specifies the use of digital signatures to provide integrity for XML elements in a SOAP message, and it specifies the use of encryption to provide confidentiality for XML elements in a SOAP message. The specification allows you to protect the body of the message or any XML elements within the body or the header. You can give different levels of protection to different elements within the SOAP message.

The advantage of using WS-Security over SSL is that it can provide *end-to-end* message-level security. This means that the message security can be protected even if the message goes through multiple services, called intermediaries.

SSL security is considered to be *point-to-point*, and the data can be decrypted prior to reaching the intended recipient.

As illustrated in Figure 2-1, if the service requester identifies itself to the intermediate gateway, and the intermediate gateway identifies itself to the service provider, the target service will normally run with the identity of the intermediate gateway rather than the service requester.

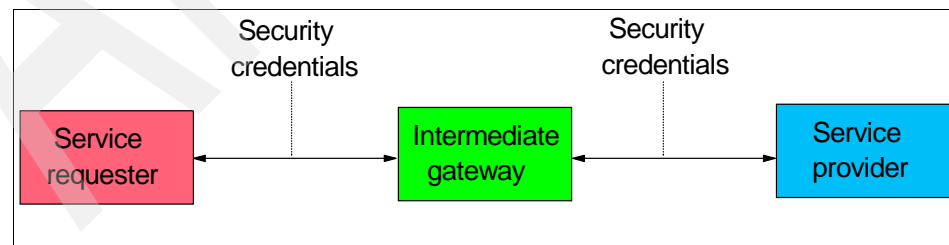


Figure 2-1 Transport-level security with an intermediate gateway

WS-Security addresses this problem by allowing security credentials to be passed within the SOAP message, so that the credentials of the service requester can be passed via an intermediate gateway, and can still be used to identify the requester to the service provider. See Figure 2-2.

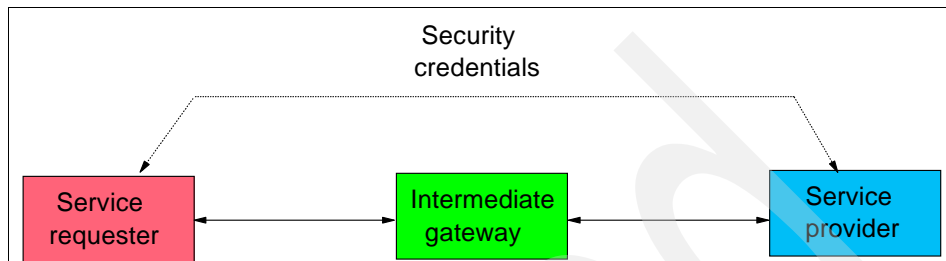


Figure 2-2 SOAP message security with an intermediate gateway

Figure 2-3 shows how a SOAP message can be extended with security data that is used to authenticate the service requester and to protect the message as it passes between the requester and the service provider. The network portion of the diagram could contain any number of intermediate nodes, some of which might not be trusted.

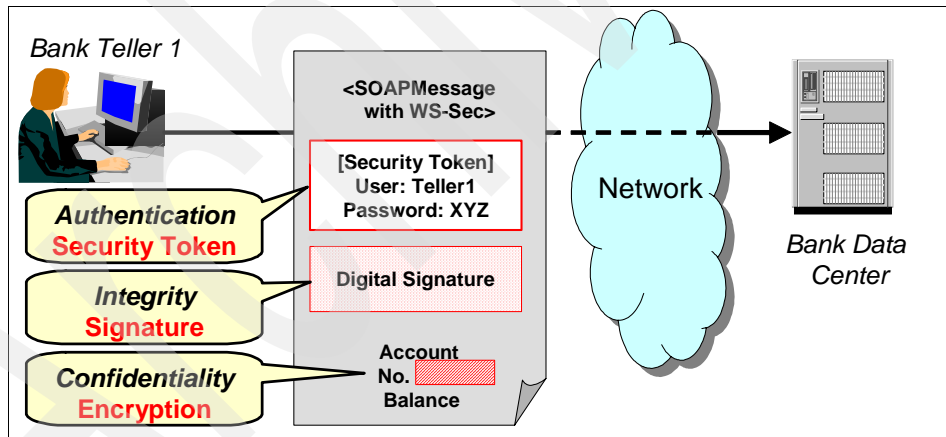


Figure 2-3 An example of a typical scenario with WS-Security

The SOAP message shown in Figure 2-3 contains three pieces of security data:

- ▶ A security token used to authenticate and identify user Teller1
- ▶ An XML digital signature to ensure that no one modifies the message while it is in transit without the modification being detected
- ▶ An account balance XML element that is encrypted to ensure confidentiality

To read more about the Web services security specifications, refer to:

- ▶ Specification: Web Services Security (WS-Security) Version 1.0 (April 2002):
<http://www.ibm.com/developerworks/webservices/library/ws-secure/>
- ▶ Web Services Security Addendum (August 2002):
<http://www.ibm.com/developerworks/webservices/library/ws-secureadd.html>
- ▶ Web Services Security: SOAP Message Security V1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>

2.1.1 Web Services-Security road map

The WS-Security specification addresses only a subset of security services. A more general security model is required to cover other security aspects such as logging and nonrepudiation. The definition of these requirements is explained in a common Web services security model framework. This road map is introduced in the following section.

2.1.2 Web Services-Security model framework

The WS-Security model introduces a set of individual interrelated specifications to form a layering approach to security. It includes several aspects of security, including identification, authentication, authorization, integrity, confidentiality, auditing, and nonrepudiation. It is based on the WS-Security specification.

The Web services security model is schematically shown in Figure 2-4.

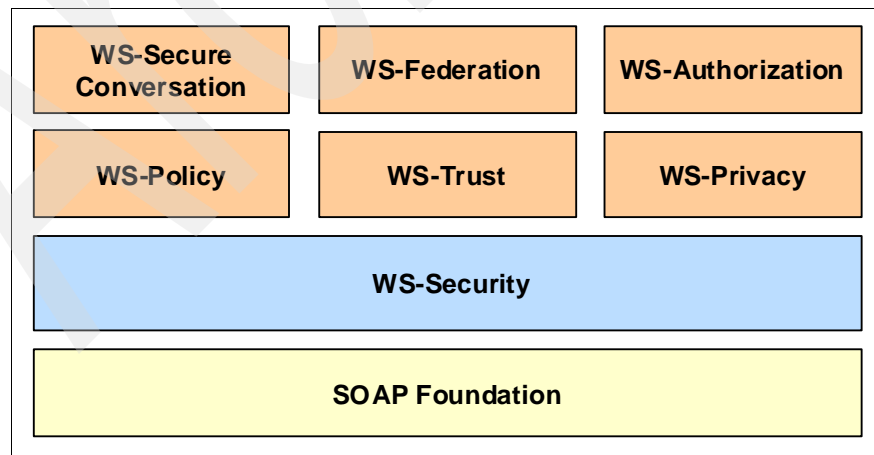


Figure 2-4 WS-Security road map

These specifications include different aspects of WS-Security:

▶ **WS-Policy**

Describes the capabilities and constraints of the security policies on intermediaries and endpoints, for example, the security tokens required, encryption algorithms supported, and privacy rules.

▶ **WS-Trust**

Describes a framework for trust models, which enables Web services to securely interoperate, manage trusts, and establish trust relationships.

▶ **WS-Privacy**

Describes a model for how Web services and requestors state privacy preferences and organizational privacy practice statements.

▶ **WS-Federation**

Describes how to manage and broker the trust relationships in a heterogeneous federated environment, including support for federated identities.

▶ **WS-Authorization**

Describes how to manage authorization data and authorization policies.

▶ **WS-Secure Conversation**

Describes how to manage and authenticate message exchanges between parties, including security context exchange and establishing and deriving session keys.

The combination of these security specifications enables many scenarios that are difficult or impossible to implement with more basic security mechanisms such as transport-level security or XML document encryption.

The first of these specifications to be supported by CICS is the WS-Trust (Web Services Trust) specification.

2.2 CICS and SOAP message security

Using WS-Security, you can apply authentication, integrity, and confidentiality at the *message* level. CICS TS V3.2 provides a security handler that supports the following security functions:

- ▶ Various mechanisms for deriving a user ID from an inbound message, including:
 - Basic authentication

- X.509 certificate
- Identity assertion
- Interoperation with a trusted third party
- ▶ Various mechanisms for attaching a security token to outbound message, including:
 - X.509 certificate
 - Identity assertion
 - Interoperation with a trusted third party
- ▶ Signature validation of inbound message signatures and signature generation for the SOAP body on outbound messages
- ▶ Decryption of encrypted data in inbound messages and encryption of the SOAP body content on outbound messages

Licensing information

The WS-Security implementation in CICS TS V3.2 contains code derived from the Apache XML Security project. The licensing terms associated with that code mean that the WS-Security implementation in CICS TS V3.2 is not licensed on the same basis as the rest of the CICS TS product.

Except for the WS-Security implementation, CICS TS V3.2 is an “ICA Program,” licensed under the relevant terms and conditions of the IBM Customer Agreement (ICA) or IBM International Customer Agreement (IICA). The WS-Security implementation is licensed under the terms and conditions of the IBM International Program License Agreement (IPLA). IPLA licensing is widely used in IBM, especially for products on distributed platforms and for one-time charge products on z/OS.

The WS-Security implementation, known as the CICS WS-Security Component, is packaged as a unique FMID with the identifier JCI650W. FMID JCI650W is licensed under the IPLA.

For this reason, the Licensed Program Specification (LPS) for CICS TS V3.2 includes a section explaining that the CICS WS-Security Component has a different licensing basis from the rest of CICS TS. Specifically, the licensing of the CICS WS-Security Component is addressed by the following three additional paper items:

- ▶ The IPLA (multi-language booklet)
- ▶ License Information (LI) document for CICS WS-Security Component (multi-language booklet)
- ▶ Proof of Entitlement for CICS WS-Security Component (multi-language sheet of paper)

2.2.1 CICS support for WS-Trust

The WS-Trust specification enhances Web Services Security by providing a framework for requesting and issuing security tokens and managing trust relationships between Web service requesters and providers. This extension to the authentication of SOAP messages enables Web services to validate and exchange security tokens of different types using a trusted third party. This third party is called a Security Token Service (STS).

For more information on WS-Trust, see the Web Services Trust Language specification that is published at:

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

Why use a Security Token Service?

A Security Token Service can be used to transform one form of user identification into another form. This is necessary when one application's security manager does not understand the user identification credentials of another. For example, CICS supports UsernameTokens, but the user identity is limited to 8 characters, while a distributed system can use user identities of 16 characters.

What is Tivoli Federated Identity Manager?

IBM Tivoli Federated Identity Manager can act as an STS by providing the necessary framework to support standards-based, federated identity management between enterprises that have established a trust relationship. Tivoli Federated Identity Manager (TFIM) provides capabilities in the areas of federated single sign-on, Web Services security management, and identity management for service-oriented architecture (SOA) components across the enterprise.

TFIM supports a wide range of security tokens, including SAML, UsernameTokens, Kerberos, LTPA, Passticket and X.509 tokens. CICS can call TFIM to validate or to issue tokens that CICS itself does not support, but which are required by Web Service partners.

For more information on TFIM, see the IBM Tivoli information center at:

<http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/index.jsp?toc=/com.ibm.tivoli.fim.doc/toc.xml>

2.2.2 SOAP message security options

There are several options available with the CICS WS-Security support, and which ones you choose will depend on the level of security required for the data and the transmission path of the data.

In this section, we look at the SOAP message security options for the following security requirements:

- ▶ Authentication
- ▶ Signing of SOAP messages
- ▶ Encrypting SOAP messages

Authentication

WS-Security provides a general purpose mechanism to associate security tokens with messages for single message authentication. It does not require you to use a specific type of security token. Instead it is designed to be extensible and support multiple security token formats to accommodate a variety of authentication mechanisms. For example, a client might provide proof of identity and proof of a particular business certification.

Example 2-1 shows a sample SOAP message without applying WS-Security. The SOAP message is an Order request for the CICS catalog example application.

Example 2-1 SOAP message without WS-Security

```
<soapenv:Envelope
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header/>
  <soapenv:Body>
    <p635:DFH0XCMN xmlns:p635="http://www.DFH0XCMN.DFH0XCP5.Request.com">
      <p635:ca_request_id>010RDR</p635:ca_request_id>
      <p635:ca_return_code>0</p635:ca_return_code>
      <p635:ca_response_message></p635:ca_response_message>
      <p635:ca_order_request>
        <p635:ca_user ID>srthstrh</p635:ca_user ID>
        <p635:ca_charge_dept>hbhhhh</p635:ca_charge_dept>
        <p635:ca_item_ref_number>10</p635:ca_item_ref_number>
        <p635:ca_quantity_req>1</p635:ca_quantity_req>
        <p635:filler1 xsi:nil="true" />
      </p635:ca_order_request>
    </p635:DFH0XCMN>
```

```
</soapenv:Body>
</soapenv:Envelope>
```

As you can see in Example 2-1, the SOAP message does not have any SOAP headers. We apply WS-Security by inserting a SOAP security header.

WS-Security defines a vocabulary that can be used inside the SOAP envelope. The XML element `<wsse:Security>` is the *container* for security-related information (*wsse* stands for *Web services security extension*).

Note: *wsse* is an arbitrary prefix. Another prefix can be used, provided that it matches the namespace definition.

When you use WS-Security for authentication, a security token is embedded in the SOAP header and is propagated from the message sender to the intended message receiver. On the receiving side, it is the responsibility of the server security handler to authenticate the security token and to set up the caller identity for the request.

In Example 2-2 we show the same SOAP message, but this time with authentication. As you can see, we have user name and password information contained in the `<UsernameToken>` element.

Example 2-2 SOAP message with WS-Security

```
<soapenv:Envelope
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1"
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.
      xsd">
      <wsse:UsernameToken>
        <wsse:Username>WEBUSER</wsse:Username>
        <wsse:Password
          Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.
          0#PasswordText">
          REDBOOKS
        </wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    <p635:DFHOXCMN xmlns:p635="http://www.DFHOXCMN.DFHOXCP5.Request.com">
```

```
<p635:ca_request_id>010RDR</p635:ca_request_id>
<p635:ca_return_code>0</p635:ca_return_code>
<p635:ca_response_message></p635:ca_response_message>
<p635:ca_order_request>
  <p635:ca_user_ID>srthstrh</p635:ca_user_ID>
  <p635:ca_charge_dept>hbhbbb</p635:ca_charge_dept>
  <p635:ca_item_ref_number>10</p635:ca_item_ref_number>
  <p635:ca_quantity_req>1</p635:ca_quantity_req>
  <p635:filler1 xsi:nil="true" />
</p635:ca_order_request>
</p635:DFH0XCMN>
</soapenv:Body>
</soapenv:Envelope>
```

The <UsernameToken> element of the SOAP message in Example 2-2 contains credentials that can be used to authenticate the user WEBUSER.

The simplest form of security token is the UsernameToken, which is used to provide a user name and (optional) password. A signed security token is one that is cryptographically signed by a specific authority. For example, an X.509 certificate is a signed security token.

Security token usage for WS-Security is defined in separate profiles such as the Username token profile and the X.509 token profile.

To read more about these security token standards, refer to:

- ▶ Web Services Security: UsernameToken Profile V1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf>
- ▶ Web Services Security: X.509 Token Profile V1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf>

The authentication options that you can choose from when implementing SOAP message authentication with CICS are as follows:

- ▶ **Basic authentication**

In service provider mode, CICS can accept a UsernameToken in the SOAP message header for authentication on inbound SOAP messages. The UsernameToken contains a Username element and a Password element. CICS verifies the Username and Password using an external security manager such as RACF. If this is successful, CICS places the Username in container DFHWS-USERID and processes the SOAP message in the pipeline. If CICS is unable to verify the UsernameToken, it returns a SOAP fault message to the service requester.

Username tokens that contain passwords are not supported on outbound SOAP messages when CICS is the service requester.

Recommendation: In CICS TS V3.1, internal tests showed that the performance of a user-written message handler program for processing a UsernameToken was significantly better than the performance achieved with the CICS-supplied security handler. In CICS TS V3.2, however, we recommend that you use the CICS-supplied security handler for basic authentication and trusted authentication because the performance of processing UsernameTokens has been significantly improved.

► **X.509 certificate authentication**

An X.509 certificate that is used for signing (see “Signing of SOAP messages” on page 42) can also be used for authentication.

In service provider mode, CICS maps the certificate to a RACF user ID and places the user ID in the container DFHWS-USERID. In service requester mode, CICS can send an X.509 certificate in the SOAP message header to the service provider that is then used for authentication purposes. The certificate is identified with the <certificate_label> element in the pipeline configuration file.

► **Trusted authentication**

In service provider pipelines, CICS can accept a UsernameToken or X.509 certificate in the SOAP message header as trusted. When a UsernameToken is used, the password is not required. When an X.509 certificate is used the certificate is mapped to a RACF user ID. CICS trusts the provided identity and places the user ID in container DFHWS-USERID.

This option is a form of *identity assertion* in which an already authenticated user identity is forwarded to CICS and CICS accepts the identity without having to re-authenticate.

In service requester pipelines, CICS can send a username token without the password in the SOAP message header to the service provider. The user ID placed in the identity token is the contents of the DFHWS-USERID container. CICS can also send an X.509 certificate. The certificate is identified with the <certificate_label> element in the pipeline configuration file.

For more information on the options for identity assertion, see “Identity assertion” on page 58.

► **Advanced authentication**

In service provider and requester pipelines, you can verify or exchange security tokens with a Security Token Service (STS) for authentication purposes. The STS enables CICS to accept and send messages that have

security tokens in the message header that are not normally supported; for example, LTPA and Kerberos tokens or SAML assertions.

For an inbound message, you can select to verify or exchange a security token. If the request is to exchange the security token, CICS must receive a username token back from the STS. For an outbound message, you can only exchange a username token for a security token.

We document a security scenario combining the capabilities of CICS and Tivoli Federated Identity Manager (TFIM) acting as an STS in Chapter 10, “Enabling WS-Trust with TFIM” on page 313.

Signing of SOAP messages

Integrity is applied to a SOAP message to ensure that no one illegally modifies the message while it is in transit. Essentially, integrity is provided by generating an XML digital signature on the contents of the SOAP message. If the message data changes illegally, the signature would no longer be valid.

Example 2-3 shows a sample SOAP message with integrity. Here, the message body part is signed and added to the SOAP security header as signature information.

Example 2-3 SOAP message with integrity

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsu:Timestamp
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
      security-utility-1.0.xsd">
      <wsu:Created>2004-11-26T09:32:31.759Z</wsu:Created>
    </wsu:Timestamp>
    <wsse:Security soapenv:mustUnderstand="1"
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
      wssecurity-sectext-1.0.xsd">
      <wsse:BinarySecurityToken
        EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
        wss-soap-message-security-1.0#Base64Binary"
        ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x5
        09-token-profile-1.0#X509"
        wsu:Id="x509bst_1080660497650146620"
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
        security-utility-1.0.xsd">
        MIIBzCCATigAwIBAgIEQZnQ7DANBgkqhkiG9w0BAQQFADAsMQswCQYDVQQGEwJVUzEMM
        AoGA1UECxMDSUJNMQ8wDQYDVQQDEwZDbG11bnQwHhcNMMDQxMTE2MTAwNTMyWhcNMDUwMj
        EOMTAwNTMyWjAsMQswCQYDVQQGEwJVUzEMMAoGA1UECxMDSUJNMQ8wDQYDVQQDEwZDbG1
```

```

1bnQwgZ8wDQYJKoZIhvcNAQEBBQADgYOAAMIGJAoGBALrQpIVkTwPc72jTzST3d+fKTYLO
qMxq6YvfFweo6ZOH7r3+jAZu880Z7Ru6iMFvajpxrRjshesPnp66binS5vQqfmPAUxyh
k+IMUAoFZQvG4byEwzDPmymys7M8Q4uZfUK7R/6PocAtwVCssx6zGsNcaaDkGYDC/5qK8+
95y4ofAgMBAAEwDQYJKoZIhvcNAQEEBQADgYEArrrFUwSVHvWt05eVBImRu8t3cBbPd1Q
ruBOuCHRnHOg9TwoCJE2JTeZV7+bCjfb17sD8K3mu/WiV20FoTzWxEgNbxzHQ5aJb7ZC
GQZ+ffc1LCxWj+pG2Eg+BbWR2xStH3NJgPUPmpieif6f5fkht16e+CDN9XXYMLqiYhR9FK
dI=
</wsse:BinarySecurityToken>
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:SignedInfo>
    <ds:CanonicalizationMethod
      Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      <ec:InclusiveNamespaces
        PrefixList="wsse ds xsi soapenc xsd soapenv "
        xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" />
      </ds:CanonicalizationMethod>
    <ds:SignatureMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
    <ds:Reference
      URI="#wssecurity_signature_id_7018050865908551142">
      <ds:Transforms>
        <ds:Transform
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
          <ec:InclusiveNamespaces
            PrefixList="xsi soapenc p821 xsd wsu soapenv "
            xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" />
          </ds:Transform>
        </ds:Transforms>
      <ds:DigestMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <ds:DigestValue>53Ed8o+4P7XquUGuRvm50AbQ4XY=</ds:DigestValue>
    </ds:Reference>
  </ds:SignedInfo>
  <ds:SignatureValue>
    SbhHeGPrsoyxXbTPdIEcybfqvoEdZ1KiYjjvWZL/dvgqMS6/oi0cdR2Di08VNombj
    Hf9h/EAov+/zvt8i5enw5AziecKr6atLVNG4jKbuNORAhts242bBfybUks4YzduoW
    YcDU9EIXUCMTjRiMbWVvvgc1k4VhTUmKb3jN+yeRA=
  </ds:SignatureValue>
  <ds:KeyInfo>
    <wsse:SecurityTokenReference>
      <wsse:Reference URI="#x509bst_1080660497650146620"
        ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-
        200401-wss-x509-token-profile-1.0#X509" />
    </wsse:SecurityTokenReference>
  </ds:KeyInfo>
</ds:Signature>
</wsse:Security>
</soapenv:Header>
<soapenv:Body

```

```
wsu:Id="wssecurity_signature_id_7018050865908551142"  
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssec  
urity-utility-1.0.xsd">  
<p821:getDayForecast xmlns:p821="http://bean.itso">  
  <theDate>2004-11-25T15:00:00.000Z</theDate>  
</p821:getDayForecast>  
</soapenv:Body>  
</soapenv:Envelope>
```

A signature is created based on a key that the sender is authorized to have. Unauthorized sniffers do not have this key. When the receiver gets the message, it too creates a signature using the message contents. Only if the two signatures match does the receiver honor the message. If the signatures are different, a SOAP fault is returned to the sender.

For inbound messages, the CICS-supplied security message handler can verify the digital signature on individual elements in the SOAP <Header> and the <Body>. The security handler verifies the following elements:

- ▶ Verify signed elements it encounters in the <Header>.
- ▶ Verify signed elements in the SOAP <Body>. If the handler is configured to expect a signed body, CICS will reject with a fault any SOAP message in which the body is not signed.

For outbound messages, the security message handler can sign the SOAP <Body> only; it does not sign the <Header>. The security handler's configuration information specifies the algorithm and key used to sign the body.

One advantage of XML digital signatures over SSL/TLS is that the signature protects the data from tampering even if the message flows through intermediary servers. However, if a process that runs in an intermediary server legitimately changes the content of messages, it is important that it does not change the signed parts of the message because this will cause the signature validation to fail in CICS.

The signing of SOAP messages causes an increase in CPU usage. The CICS support for XML digital signatures is dependent on Integrated Cryptographic Service Facility (ICSF) services and therefore the configuration and startup of ICSF is a requirement for using this support. See "ICSF services used by System SSL" on page 146 for information about what hardware cryptographic devices are required for XML digital signature processing.

Important: The Integrated Cryptographic Service Facility (ICSF) must be started and configured with cryptographic devices in order to use the CICS WS-Security XML digital signature support.

We document how signed SOAP messages are processed by CICS in Chapter 7, “Signing the SOAP message” on page 221.

Encrypting SOAP messages

XML encryption is applied to a SOAP message to ensure that only the intended recipient of the message can read the message.

Example 2-4 shows a sample SOAP message with confidentiality using XML encryption. Here, the message body part is encrypted and a security header with encryption information (including information on the secret key identifier) is added.

Example 2-4 SOAP message with confidentiality

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsu:Timestamp
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-ws
        security-utility-1.0.xsd">
      <wsu:Created>2004-11-26T09:34:50.838Z</wsu:Created>
    </wsu:Timestamp>
    <wss:Security soapenv:mustUnderstand="1"
      xmlns:wss="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-w
        ssecurity-secext-1.0.xsd">
      <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
        <EncryptionMethod
          Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
          <wss:SecurityTokenReference>
            <wss:KeyIdentifier
              ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-
                200401-wss-x509-token-profile-1.0#X509v3SubjectKeyIdentif
                  ier">
              Vniy7MUOXBumPoH1MNbDpiIWOPA=
            </wss:KeyIdentifier>
          </wss:SecurityTokenReference>
        </ds:KeyInfo>
      <CipherData>
        <CipherValue>
          0+2mTsRjU1iN1wANv1kGdzpkRV1GQc5epAT3p5Eg5UNA3H3YAX5VrdgMQmj1
          wzdSZLDEzBtcHPJq3c8c0AgmAy9EVdgcXIn/Zev+80jMDn/HN2HfodyjURtIY
          Bg480SSkot0fy+YpBSXNR/MTfs1HT2H/Mjw/CyIbomWdQZHM=
        </CipherValue>
      </CipherData>
      <ReferenceList>
```

```

        <DataReference
          URI="#wssecurity_encryption_id_6866950837840688804"/>
      </ReferenceList>
    </EncryptedKey>
  </wsse:Security>
</soapenv:Header>
<soapenv:Body>
  <EncryptedData
    Id="wssecurity_encryption_id_6866950837840688804"
    Type="http://www.w3.org/2001/04/xmlenc#Content"
    xmlns="http://www.w3.org/2001/04/xmlenc#">
    <EncryptionMethod
      Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
    <CipherData>
      <CipherValue>
        OvLek01buZhFB11BNL4Kos195YHwYw0kSbMxkbI2pk7n117g0prPS2Ba2hyrXHABGQVmo
        sWpgqt+zijCPHUQCMmm3qgFraK11DPMmwP94Hvgx1gBmPw1Unt+WM4aKLNrHDnwwcQX5
        R07KT+fhFp4wxFEABwfHqzvTGNK3xRwJE=
      </CipherValue>
    </CipherData>
  </EncryptedData>
</soapenv:Body>
</soapenv:Envelope>

```

Confidentiality is provided by encrypting the contents of the SOAP message using XML encryption. If the SOAP message is encrypted, only a service that knows the key for confidentiality can decrypt and read the message.

CICS provides support for encrypting the SOAP message body using either the Triple DES algorithm or the AES algorithm. The secret key is included in the message and is itself encrypted using the intended recipient's public key with the asymmetric key encryption algorithm RSA 1.5. See Chapter 3, "Elements of cryptography" on page 69 for an explanation of the difference between secret key cryptography and public key cryptography.

For inbound messages, the CICS-supplied security message handler can decrypt individual elements in the SOAP <Body>, and can decrypt elements in the SOAP <Header> if the SOAP body is also encrypted.

The security message handler always decrypts the following elements:

- ▶ Elements it encounters in the <Header> in the order that the elements are found.
- ▶ Elements in the SOAP <Body>. If the handler is configured to expect an encrypted body, CICS will reject with a fault any SOAP message in which the body is not encrypted.

For outbound messages, the security message handler supports encryption of the contents of the SOAP <Body> only; it does not encrypt any elements in the <Header>. When the security message handler encrypts the <Body>, all elements in the body are encrypted with the same algorithm and using the same key. The algorithm and information about the key are specified in the handler's configuration information.

One advantage of XML encryption over SSL/TLS is that the encrypted message remains confidential even if the message flows through intermediary servers. SSL/TLS assures the security of the message during transmission, but if the messages can be received and forwarded by intermediates, secure end-to-end communication cannot be guaranteed.

XML encryption also allows specific sensitive parts of the message to be encrypted rather than having to encrypt the whole message. One important consideration when intermediaries are used, however, is that XML encryption might make it difficult to perform content-based routing of SOAP messages because the intermediary server will not be able to read all parts of the message body.

XML encryption of SOAP messages causes an increase in CPU usage. The CICS support for XML encryption is dependent on Integrated Cryptographic Service Facility (ICSF) services and therefore the configuration and startup of ICSF is a requirement for using this support. See "ICSF" on page 138 for information about what hardware cryptographic devices are required for XML encryption processing.

Important: The Integrated Cryptographic Service Facility (ICSF) must be started and configured with cryptographic devices in order to use the CICS WS-Security XML encryption support.

The Redbooks publication, *Implementing CICS Web Services*, SG24-7206 contains a step-by-step guide on how to configure CICS for processing XML encrypted SOAP messages.

You can choose to both sign and encrypt a SOAP message. This provides both message integrity and confidentiality.

2.2.3 Enabling CICS for WS-Security processing

To enable your CICS TS V3.2 region for the full range of WS-Security processing, you must apply a number of updates to your CICS region.

- ▶ Install the free IBM XML Toolkit for z/OS v1.9. You can download it from the following site, and you must install version 1.9:

<http://www.ibm.com/servers/eserver/zseries/software/xml/>

Important: Later versions of the IBM XML Toolkit do not work with Web Services Security support in CICS.

- ▶ Apply ICSF APAR OA14956 if it is not already installed in your CICS region.
- ▶ Add the following libraries to the DFHRPL concatenation:
 - hlq.SIXMLOD1
 - hlq.SCEERUN
 - hlq.SDFHWSLD

Where hlq is the high-level qualifier of the CICS region. The first two libraries contain DLLs that the security handler requires at run time:

- The XML toolkit provides IXM4C56 in hlq.SIXMLOD1.
- The Language Environment® runtime provides C128N in hlq.SCEERUN.

The hlq.SDFHWSLD library enables CICS to find the DFHWSSE1 and DFHWSXXX Web Services Security modules.

- ▶ You might have to increase the value of the EDSALIM system initialization parameter. The three DLLs that must be loaded require approximately 15 MB of EDSA storage.

Note: These prerequisites are not necessary for *all* CICS WS-Security scenarios. Refer to the security scenario chapters for specific information on implementation prerequisites.

2.2.4 Configuring the CICS-supplied security handler

To implement WS-Security in CICS TS for either a service provider or a service requester, you must include a `<wsse_handler>` element in the configuration file for the appropriate pipeline. Use the following sub-elements of `<wsse_handler>` to provide configuration information to the CICS-supplied security handler.

1. Either of the following elements:
 - An optional `<authentication>` element.
 - In a service requester pipeline, the `<authentication>` element specifies the type of authentication that the security header of outbound SOAP messages will use.

- In a service provider pipeline, the element specifies whether CICS will use the security tokens in an inbound SOAP message to determine the user ID under which work will be processed.

The <authentication> element has two attributes: trust and mode. These attributes determine whether asserted identity is used and the combination of security tokens used in a SOAP message. The trust attribute can be set to either none, basic, or signature. The mode attribute can also be set to either none, basic, or signature.

A trust token, for example, could be a UsernameToken or an X.509 token. The trust token is used to check that the sender has the correct permissions to assert identities, and the identity token holds the asserted identity (user ID) under which the request is to run.

Note: If you use asserted identity, it requires that the service provider trusts the requester to make this assertion. In CICS, the trust relationship is established with security manager surrogate definitions: the requesting identity must have the correct authority to start work on behalf of the asserted identity.

For more information about the meaning and valid combinations of the trust and mode attributes, refer to the CICS Transaction Server for z/OS V3.2 Web Services Guide, SG34-6838. See also the security scenarios documented in following chapters: Chapter 8, “Identity assertion with WebSphere for z/OS” on page 259 and Chapter 9, “Identity assertion with WebSphere DataPower” on page 281.

The <authentication> element can contain the following elements:

- <certificate_label>
Optional. Specifies the label associated with an X.509 digital certificate. Ignored in a service provider pipeline.
- <suppress/>
Optional. For a service provider, the handler will not use any security tokens in the message to determine under which user ID to run. For a service requester, the handler will not add to the SOAP message any of the security tokens required for authentication.
- <algorithm>
Specifies the URI of the signature algorithm. You must specify this element if the combination of trust and mode attribute values indicate that the messages are signed. You can specify one of the algorithms shown in Table 2-1.

Table 2-1 Signature algorithms for inbound SOAP messages

Algorithm	URI
Digital Signature Algorithm with Secure Hash Algorithm 1 (DSA with SHA1)	http://www.w3.org/2000/09/xmldsig#dsa-sha1
Rivest-Shamir-Adleman algorithm with Secure Hash Algorithm 1 (RSA with SHA1)	http://www.w3.org/2000/09/xmldsig#rsa-sha1

- An optional `<sts_authentication>` element.

The action attribute on this element specifies what type of request should be sent to the Security Token Service (STS): *issue* or *validate*.

- If the request is to issue an identity token, then CICS uses the values in the nested elements to request an identity token of the specified type.
- If the request is to validate an identity token, then CICS uses the values in the nested elements to request the STS to validate an identity token of the specified type.

If you specify an `<sts_authentication>` element, you must also specify an `<sts_endpoint>` element. When this element is present, CICS uses the URI in the `<endpoint>` element to send a request to the STS.

Within the `<sts_authentication>` element, code the following elements:

- An `<auth_token_type>` element. This element is required when you specify an `<sts_authentication>` element in a service requester pipeline and optional in a service provider pipeline.

In a service requester pipeline, the `<auth_token_type>` element indicates the type of token that the STS will issue when CICS sends it the user ID contained in the DFHWS-USERID container. The token that CICS receives from the STS is placed in the header of the outbound message.

In a service provider pipeline, the `<auth_token_type>` element is used to determine which identity token CICS takes from the message header and sends to the STS to exchange or validate. CICS uses the first identity token of the specified type in the message header. If you do not specify this element, CICS uses the first identity token that it finds in the message header.

- In a service provider pipeline only, an optional, empty `<suppress/>` element. If this element is specified, the handler does not attempt to use any security tokens in the message to determine under which user ID the work will run, including the identity token returned by the STS.

2. An optional `<sts_endpoint>` element. Use this element only if you have also specified an `<sts_authentication>` element.

In the `<sts_endpoint>` element, you code an `<endpoint>` element. This element contains a URI that points to the location of the STS on the network.

The STS endpoint can be specified as an HTTP endpoint or a WebSphere MQ endpoint (using the JMS format of URI).

Recommendation: We recommend that you use SSL or TLS to keep the connection to the STS secure.

3. An optional, empty `<expect_signed_body/>` element. The `<expect_signed_body/>` element indicates that the `<body>` of the inbound message must be signed. If the body of an inbound message is not correctly signed, CICS rejects the message with a security fault.
4. An optional, empty `<expect_encrypted_body/>` element. The `<expect_encrypted_body/>` element indicates that the `<body>` of the inbound message must be encrypted. If the body of an inbound message is not correctly encrypted, CICS rejects the message with a security fault.
5. An optional `<sign_body>` element. If this element is present, CICS will sign the `<body>` of the outbound message. It contains the following elements:
 - `<algorithm>`

Specifies the URI of the algorithm used to sign the body of the SOAP message.

CICS supports the following signature algorithm for outbound SOAP messages:

 - Rivest-Shamir-Adleman algorithm with Secure Hash Algorithm 1 (RSA with SHA1), which is specified using the URI <http://www.w3.org/2000/09/xmldsig#rsa-sha1>
 - `<certificate_label>`

Specifies the label associated with an X.509 digital certificate. The digital certificate should contain the private key since this was used to sign the message. The public key associated with the private key is then sent in the SOAP message, which allows the signature to be validated.
6. An optional `<encrypt_body>` element. If this element is present, CICS will encrypt the `<body>` of the outbound message. It contains the following elements:
 - `<algorithm>`

Specifies the URI identifying the algorithm used to encrypt the body of the SOAP message. CICS supports the encryption algorithms shown in Table 2-2.

Table 2-2 Encryption algorithms

Algorithm	URI
Triple DES in cipher block chaining mode	http://www.w3.org/2001/04/xmlenc#triple-des-cbc
AES with a key length of 128 bits in cipher block chaining mode	http://www.w3.org/2001/04/xmlenc#aes128-cbc
AES with a key length of 192 bits in cipher block chaining mode	http://www.w3.org/2001/04/xmlenc#aes192-cbc
AES with a key length of 256 bits in cipher block chaining mode	http://www.w3.org/2001/04/xmlenc#aes256-cbc

– <certificate_label>

Specifies the label associated with an X.509 digital certificate. The digital certificate should contain the public key of the intended recipient of the SOAP message so that it can be decrypted with the private key when the message is received.

- In provider pipelines only, an optional <reject_signature/> element. If this element is present, CICS rejects any message that includes a certificate in its header that signs part or all of the message body. A SOAP fault is issued to the Web service requester.

Recommendation: Use the <reject_signature/> element if you do not want CICS to process signed SOAP messages. This can be useful to protect against *denial of service* attacks.

- In provider pipelines only, an optional <reject_encryption/> element. If this element is present, CICS rejects any message that is partially or fully encrypted. A SOAP fault is issued to the Web service requester.

Recommendation: Use the <reject_encryption/> element if you do not want CICS to process encrypted SOAP messages. This can be useful to protect against *denial of service* attacks.

Example <wsse_handler> configuration

Example 2-5 shows an example <wsse_handler> element for a service provider pipeline which configures the CICS-supplied security handler to access an STS to issue a security token.

Example 2-5 <wsse_handler>

```
<wsse_handler>
  <dfhwsse_configuration version="1">
    <sts_authentication action="issue">
      <auth_token_type>
        <namespace>http://docs.oasis-open.org/wss/2004/01/oasis-20
          0401-wss-wssecurity-secext-1.0.xsd</namespace>
        <element>UsernameToken</element>
      </auth_token_type>
    </sts_authentication>
    <sts_endpoint>
      <endpoint>http://mett.pdl.pok.ibm.com:9610/TrustServer/Security
        TokenService</endpoint>
    </sts_endpoint>
  </dfhwsse_configuration>
</wsse_handler>
```

The <wsse_handler> element contains a <dfhwsse_configuration> element that specifies configuration information for the security handler.

- ▶ In the <sts_authentication> element, the action="issue" attribute specifies that CICS is to request a security token from the STS.
- ▶ The <auth_token_type> specifies which identity token CICS takes from the SOAP Header and sends to the STS (in this case, a UsernameToken).
- ▶ The <sts_endpoint> element contains one child element, <endpoint>, which contains the URI that points to the location of the STS on the network. This is an HTTPS endpoint.

For more detailed information on configuring CICS to use an STS, see “Configuring CICS for WS-Trust processing” on page 321. For other examples of pipeline configurations using the WS-Security support, refer to the other security scenario chapters in this book.

2.2.5 Custom security handlers

You might want to use your own security procedures and processing; if so, you can write a custom security handler to process secure SOAP messages.

You must decide the level of security that your security handler will support and ensure that an appropriate SOAP fault is returned when a message includes security that is not supported.

Here is a likely set of steps that your security handler would implement:

1. Retrieve the DFHREQUEST or DFHRESPONSE container using an EXEC CICS GET CONTAINER command.
2. Parse the XML to find the security token that is in the WS-Security message header. The header starts with the `<wsse:Security>` element.

The security token might be a user name and password, a digital certificate, or an encryption key. A message can have many tokens in the security header, so your handler must identify the correct one to process.

3. Perform the appropriate processing, depending on what security is implemented in the message.
 - If you want to perform basic authentication, issue an EXEC CICS VERIFY PASSWORD command. This command checks the user name and password in the security header of the message. If this command is successful, update the DFHWS-USERID container with an EXEC CICS PUT CONTAINER. Otherwise, issue an EXEC CICS SOAPFAULT CREATE command.

You might also want to write an audit record each time a service is requested, for example, you could write a message to a CICS user journal.

- If you want to perform advanced authentication, either by exchanging or validating a range of tokens with an STS, use the CICS provided Trust client interface which enables you to interact with the STS directly.

The Trust client interface is an enhancement to the CICS-supplied program DFHPIRT. This program is normally used to start a pipeline when a Web service requester application has not been deployed using the CICS Web services assistant. Its functionality has now been extended so that it can act as the Trust client interface to the STS.

You can invoke the Trust client interface by linking to DFHPIRT from a message handler, passing a channel called DFHWSTC-V1 and a set of security containers. Using these containers, you have the flexibility to request either a validate or issue action from the STS, select what token type to exchange, and pass the appropriate token from the message header.

DFHPIRT dynamically creates a pipeline, composes a Web service request from the security containers, and sends it to the STS. DFHPIRT waits for the response from the STS and passes this back in the DFHWS-RESTOKEN container to the message handler.

If the STS encounters an error, it returns a SOAP fault. DFHPIRT puts the fault in the DFHWS-STSFault container and returns to the linking program in the pipeline.

You must define your security handler program in CICS and update the pipeline configuration file, ensuring that it is correctly placed in the XML.

In a service requester pipeline configuration file, the security handler should be configured to run at the end of the pipeline. In a service provider pipeline configuration file, configure the security handler to run at the beginning of the pipeline.

For an example of a custom security handler, see the book, *Implementing CICS Web Services*, SG24-7206.

2.3 WebSphere and SOAP message security

WebSphere Application Server V6.1 is based on the implementation of WS-Security in the following OASIS specification and profiles:

- ▶ WS-I Basic Security Profile
<http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html>
- ▶ Web Services Security: SOAP Message Security 1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>
- ▶ Web Services Security: UsernameToken Profile 1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf>
- ▶ Web Services Security: X.509 Certificate Token Profile V1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf>

WebSphere Application Server uses the J2EE™ 1.4 Web services deployment model to implement WS-Security. The Web services security constraints are specified in the IBM extension of the Web services deployment descriptors and bindings. The Web services security runtime enforces the security constraints specified in the deployment descriptors. One of the advantages of this deployment model is that you can define the Web services security requirements outside of the application business logic. With the separation of roles, the application developer can focus on the business logic, and the security expert can specify the security requirement.

Figure 2-5 shows the high-level architecture model that is used to secure Web services in WebSphere Application Server.

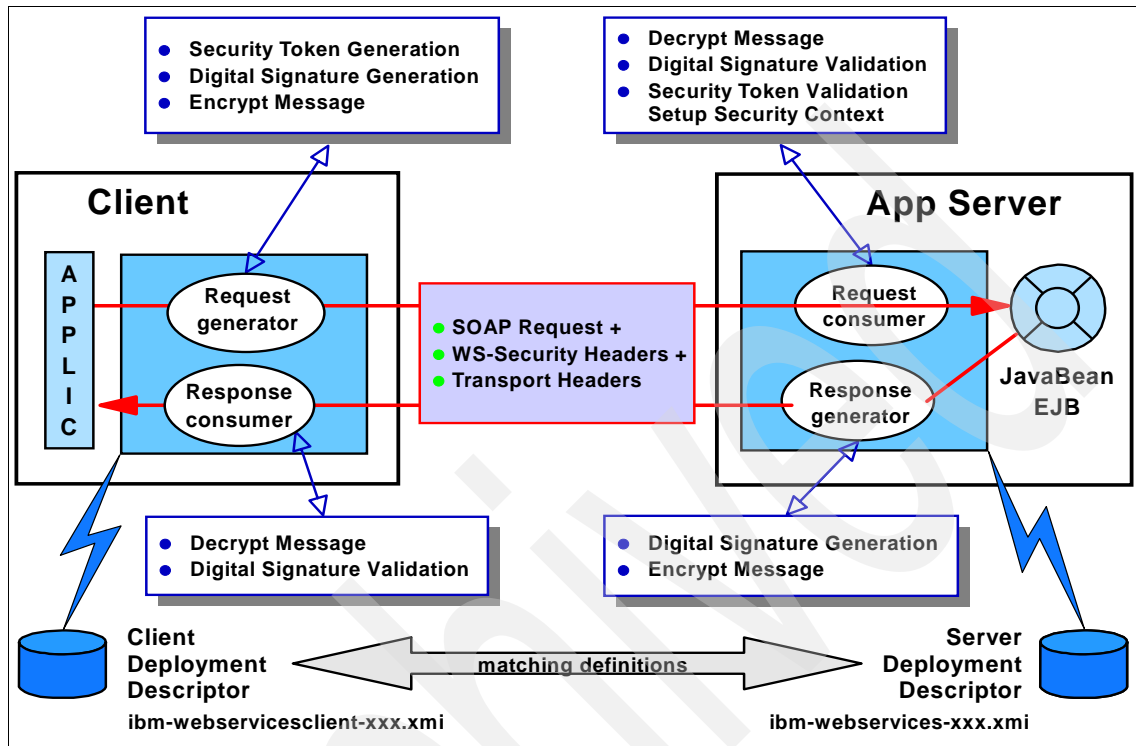


Figure 2-5 WebSphere Application Server support for WS-Security

As shown in the figure, there are two sets of configurations on both the client side and the server side.

► **Request generator**

This client-side configuration defines the Web services security requirements for the outgoing SOAP message request. These requirements might involve generating a SOAP message request that uses a digital signature, incorporates encryption, and attaches a security token.

► **Request consumer**

This server-side configuration defines the Web services security requirements for the incoming SOAP message request. These requirements might involve verifying that the required integrity parts are digitally signed, verifying the digital signature, verifying that the required confidential parts were encrypted by the request generator, decrypting the required confidential parts, validating the security token, and verifying that the security context is set up with the appropriate identity.

▶ **Response generator**

This server-side configuration defines the Web services security requirements for the outgoing SOAP message response. These requirements might involve generating the SOAP message response with Web services security, including digital signature; and encrypting and attaching a security token, if necessary.

▶ **Response consumer**

This client-side configuration defines the Web services security requirements for the incoming SOAP response. The requirements might involve verifying that the integrity parts are signed and the signature is verified; verifying that the required confidential parts are encrypted and that the parts are decrypted; and validating the security token, if necessary.

The Web services security requirements that are defined in the request generator must match the request consumer. The requirements that are defined in the response generator must match the response consumer. Otherwise, the request or response is rejected because the Web services security constraints cannot be met by the request consumer and response consumer.

The format of the Web services security deployment descriptors and bindings are IBM proprietary. However, the following tools are available to edit the deployment descriptors and bindings:

- ▶ Rational® Application Developer (RAD)
- ▶ Application Server Toolkit (AST)

RAD and AST provide two options to configure WS-Security:

- ▶ Enabling Web services security using the *WS Security wizard*.

You can enable production-level security for your Web services using the WS Security wizard, which provides the following options:

- Add an XML digital signature
- Add XML encryption
- Add a stand alone security token

For an example of using the WS Security wizard, see “Configuring the request generator for signing” on page 237.

- ▶ Enabling Web services security manually

If the WS Security wizard does not meet your needs, you have the option of manually securing Web services. This process gives you more control over the security settings, but is more time consuming and error prone than enabling security using the wizards.

You can use the Web Services editor to configure the server side security and the Web Services Client editor to configure the client side security. Or, you can deal with the deployment descriptor files directly, such as `ibm-webservices-bnd.xmi` and `ibm-webservices-ext.xmi` for the server side configuration, and `ibm-webservicesclient-bnd.xmi` and `ibm-webservicesclient-ext.xmi` for the client side configuration.

For an example of using the Web Services editor, see “Configuring the request generator for identity assertion” on page 265.

2.4 Identity assertion

Identity assertion is an authentication mechanism that is applied among three parties: a client, an intermediary server, and a target server:

- ▶ A request message is sent to an intermediary server with a client's security token.
- ▶ The intermediary server (for example, WebSphere Application Server or a DataPower SOA appliance) authenticates the client and transfers the client's request message and identity to the target server with the *intermediary's* security token.

Identity assertion is an extended security mechanism supported by CICS TS and WebSphere Application Server. There are several options for sending the client's identity with the intermediary's token to the target server.

It is possible to use identity assertion to secure a CICS Web service via an intermediary server. For example, we could configure the intermediary server as follows:

- ▶ Define a security requirement for the request consumer such that the requester must provide a binary security token, such as an X.509 certificate.
- ▶ Map the certificate to a user identity.
- ▶ Define a security requirement for the request generator such that the intermediary server propagates the service requester's identity to CICS in a `UsernameToken`.

The intermediary server must establish a trust relationship with the CICS region by authenticating itself and then by being recognized as a trusted partner of the CICS region. CICS supports two different models:

- Trust token** The intermediary server sends a trust token to CICS.
- Blind trust** Trust is established at the transport level rather than at the SOAP message level.

2.4.1 Trust token model

In this model, SOAP messages that flow between the intermediary server and CICS contain a trust token and an identity token. The trust token is used to check that the sender has the correct permissions to assert identities, and the identity token holds the asserted identity (user ID) under which the request is to run. When using the CICS-provided support for WS-Security, a trust token can be a UsernameToken or an X.509 token.

The advantage of the trust token model is that it is independent of the mechanism used to transport the SOAP messages. The disadvantage, however, is the overhead of validating the trust token for each SOAP request.

2.4.2 Blind trust model

In this model, SOAP messages that flow between the intermediary server and CICS contain only an identity token. The trust relationship between the intermediary server and CICS must be established using a transport-based mechanism such as SSL client authentication.

The advantage of the blind trust model is that the trust established between the intermediary server and CICS can be persistent (for example, by using SSL persistent connections) and does not need to be re-established for each SOAP message. The disadvantage, however, is that it is dependent on the mechanism used to transport the SOAP messages.

We document an identity assertion scenario using the blind trust model in the following chapters: Chapter 8, “Identity assertion with WebSphere for z/OS” on page 259 and Chapter 9, “Identity assertion with WebSphere DataPower” on page 281.

Recommendation: The blind trust model offers significant performance advantages over the trust token model.

2.5 WebSphere DataPower SOA appliances

IBM WebSphere DataPower SOA appliances are purpose-built, easy-to-deploy network devices to simplify and accelerate XML and Web services deployments.

The DataPower family contains rack-mountable network devices that overcome many of the challenges that face SOA implementations today, including security threats. At a high level, DataPower appliances offer the following features:

- ▶ 1U (1.75-inch thick) rack-mountable, purpose-built network appliances

- ▶ XML/SOAP firewall, field-level XML security, data validation, XML Web services access control, and service virtualization
- ▶ Lightweight and protocol-independent message brokering, integrated message-level security and fine-grained access control, and the ability to bridge important transaction networks to SOAs and ESBs
- ▶ High performance, multi-step, wire-speed message processing, including XML, XML Stylesheet Language Transformation (XSLT), XPath, and XML Schema Definition (XSD)
- ▶ Centralized Web services policy and service-level management
- ▶ Web services (WS) standard support, such as WS-Security, Security Assertion Markup Language (SAML) 1.0/1.1/2.0, portions of the Liberty Alliance protocol, WS-Federation, WS-Trust, XML Key Management Specification (XKMS), Radius, XML Digital Signature, XML-Encryption, Web Services Distributed Management (WSDM), WS-SecureConversation, WS-Policy, WS-SecurityPolicy, WS-ReliableMessaging, SOAP, Web Services Description Language (WSDL), Universal Description, Discovery, and Integration (UDDI)
- ▶ Transport layer flexibility, which supports HTTP/HTTPS, MQ, Secure Sockets Layer (SSL), File Transfer Protocol (FTP), and others
- ▶ Scalable, wire-speed, any-to-any message transformation, such as arbitrary binary, flat text and XML messages, which include COBOL copybook, CORBA, CICS, ISO 8583, ASN.1, EDI, and others

There are three types of DataPower appliance available, each building on the features of the last:

- ▶ IBM WebSphere DataPower XML Accelerator XA35
Accelerates common types of XML processing by offloading this processing from servers and networks. It can perform XML parsing, XML Schema validation, XPath routing, Extensible Stylesheet Language Transformations (XSLT), XML compression, and other essential XML processing with wirespeed XML performance.
- ▶ IBM WebSphere DataPower XML Security Gateway XS40
Provides a security-enforcement point for XML and Web services transactions, including encryption, firewall filtering, digital signatures, schema validation, WS-Security, XML access control, XPath, and detailed logging.
- ▶ IBM WebSphere DataPower Integration Appliance XI50
Transport-independent transformations between binary, flat text files, and XML message formats. Visual tools are used to describe data formats, create mappings between different formats, and define message choreography.

For full product information about IBM WebSphere DataPower SOA Appliances, see:

<http://www.ibm.com/software/integration/datapower/index.html>

A DataPower SOA appliance can be used in conjunction with CICS Web services to help secure the services and to offload expensive operations by processing the complex part of XML messages (such as an XML digital signature) at wirespeed (see Figure 2-6).

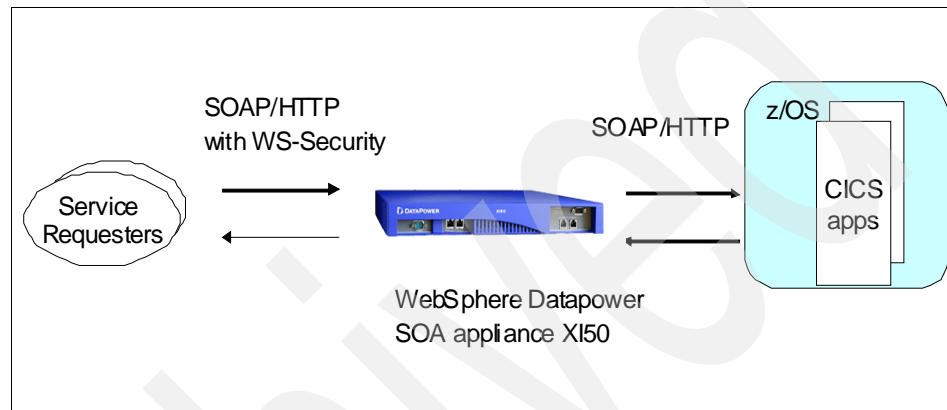


Figure 2-6 Using a WebSphere DataPower SOA Appliance with CICS Web services

We document a security scenario combining the capabilities of WebSphere DataPower and CICS in “Identity assertion with WebSphere DataPower” on page 281.

2.6 Comparison of transport level and SOAP message security

We have shown in this chapter that it is possible to implement Web services security at two levels: the transport level and the SOAP message level. If your Web services environment is simple (for example, it does not span multiple nodes), a security solution based on transport-level security alone might be all that you require. For more complex scenarios, however, it might not be enough on its own.

In this section, we provide general guidelines to help you decide what type of security solution to implement.

- ▶ You might choose to use only transport-level security to secure your CICS Web services environment when:
 - No intermediaries are used in the Web service environment or, if there are intermediaries, then you can guarantee that once the data is decrypted, it cannot be accessed by an untrusted node or process.
 - The transport is only based on HTTP.
 - Performance is your primary concern.

SSL/TLS is a mature technology that has been optimized over a long period of time, and there are ways of optimizing performance such as persistent TCP/IP connections and SSL session ID reuse. These optimizations mean that expensive security functions, such as SSL handshaking, can be avoided for service requests following the initial handshake.

WS-Security support, in comparison, is completely stateless, and expensive security functions, such as XML digital signature validation, are repeated for each service request.
 - The Web services client is a stand-alone Java™ program.

WS-Security can only be applied to clients that run in a Web services environment that supports the WS-Security specification (for example, WebSphere Application Server).
- ▶ You might choose to use WS-Security (possibly in addition to transport-level security) when:
 - Intermediaries are used, some of which might be untrusted.

Security credentials that flow in the SOAP message can pass through any number of intermediaries. Protecting confidential information in the actual SOAP message can avoid the overhead of encrypting and decrypting via SSL at every intermediary node.

Furthermore, an intermediary might be able to provide an authentication service to CICS, such that the intermediary server authenticates the Web service client and then flows an *asserted* identity to CICS.
 - Multiple transport protocols are used.

WS-Security works across multiple transports and is independent of the underlying transport protocol.
 - The Web service partners support WS-Security and a general decision has been taken to flow security tokens in accordance with the WS-Security specification.
 - You might choose to implement your own security procedures and processing by writing a custom message handler program that can process secure SOAP messages in the pipeline.

2.7 Planning considerations

The following list of questions and comments will help you to choose between the different options available for securing CICS Web services. First, we outline the main planning considerations for the following security requirements:

- ▶ Authentication
- ▶ Authorization
- ▶ Confidentiality
- ▶ Data integrity

We then look at the main performance considerations.

Authentication

Consider the following authentication related questions:

- ▶ Does the service requester need to authenticate?

This can be decided for specific services rather than a general rule for the application or for the CICS region. It might be appropriate to run read-only services using a generic user ID, whereas more sensitive services might need the requester to authenticate. This split can be made by running secured services on a different pipeline to unsecured services.

- ▶ Will you use transport based or SOAP message based authentication?

You might choose to use transport based authentication when:

- No intermediaries are used, that is, the service requester sends the request directly to the service provider.
- A single transport is used, for example, HTTP.
- Performance is your primary concern (see “Performance” on page 66).
- The Web services client is a stand-alone Java program and does not support WS-Security.
- Web services atomic transactions (WS-AT) are being processed by this pipeline.

Note: CICS does not support WS-Security or WS-Trust in pipelines that are used for atomic transactions.

You might choose to use SOAP message based authentication when:

- Intermediaries are used.

Security credentials that flow in the SOAP message can pass through any number of intermediaries.

- Multiple transport protocols are used.
When using an asynchronous transport such as WebSphere MQ, be aware that certain security tokens, for example, passtickets and LTPA tokens, become invalid after an expiration timeout period.
- Your Web service partners support WS-Security and a general decision has been taken to flow security tokens in accordance with the WS-Security specification.
- You choose to implement your own security procedures and processing by writing a custom message handler program that can process secure SOAP messages in the pipeline.
- ▶ What token type is being used?
 - UsernameTokens or X.509 certificates should be processed directly by the CICS-supplied security handler.
 - You will most likely have to configure the CICS-supplied handler to call an STS if other token types are used.
 - You can also use an STS to process non-standard token types or you can write a custom security handler.
- ▶ Who authenticates the service requester, CICS or an intermediary server?
CICS can authenticate service requesters directly, or an intermediary might be able to provide an authentication service to CICS. In this case, the intermediary server authenticates the service requester and then flows an *asserted* identity to CICS.

Authorization

Consider the following authorization related questions:

- ▶ Does the CICS task have to run with the service requester's identity?
If it is not necessary to run the CICS task with the service requester's identity, a RACF user ID can be specified in a URIMAP (this avoids running the Web service with the CICS default user ID).
If it is necessary to run the CICS task with the service requester's identity:
 - In the case where no intermediaries are used, CICS authorization processing can be based on the RACF user ID that is associated with the security token that is used by the requester to authenticate.
 - In the case where intermediaries are used, CICS authorization processing can be based on the asserted identity token that is passed by the intermediary.

Surrogate authorization checking should be used to ensure that the intermediary has the correct authority to start work on behalf of the asserted identity.

Confidentiality

Consider the following confidentiality related questions:

- ▶ Does the sensitivity of the data warrant encryption?

If SOAP messages do not contain sensitive data, or if the messages are only transmitted within an internal secure network, then it might be reasonable to flow unencrypted messages.

A single CICS region can process encrypted and unencrypted messages. You can define a TCPIP SERVICE for HTTPS and a TCPIP SERVICE for HTTP. You can also define pipelines that expect to receive XML encrypted messages and other pipelines that will reject such messages.

- ▶ Are intermediaries used?

SSL/TLS only provides privacy of the data during the message transmission. Protecting confidential information in the actual SOAP message using XML encryption might be necessary in order to protect message confidentiality within every intermediary node.

Be aware that XML encryption might make it difficult to perform content-based routing of SOAP messages because the intermediary server will not be able to read all parts of the message body.

- ▶ Does CICS need to deal with encrypted messages?

It might be appropriate for an intermediary server to terminate an HTTPS session and forward the request to CICS across a secured network as an HTTP request.

Equally, it might be appropriate for an intermediary server to decrypt XML encrypted messages and forward an unencrypted message to CICS.

- ▶ Does CICS call an STS?

You should use SSL/TLS to keep the connection to the STS secure.

Data integrity

Consider the following data integrity related questions:

- ▶ Does the integrity of the data warrant protection?

If SOAP messages do not contain critical data, or if the messages are only transmitted within an internal secure network, then it might be reasonable to flow unsigned messages.

A single CICS region can process signed and unsigned messages, for example, you can define pipelines that expect to receive signed messages and other pipelines that will reject signed messages.

- ▶ Are intermediaries used?

SSL/TLS only provides integrity of the data during the message transmission. XML signatures might be necessary in order to protect message integrity within every intermediary node.

Be aware that the intermediary server does not change the signed parts of a message because this will cause the signature validation to fail in CICS.

- ▶ Does CICS need to deal with signed messages?

It might be appropriate for an intermediary server to terminate an HTTPS session and forward the request to CICS across a secured network as an HTTP request.

Equally, it might be appropriate for an intermediary server to validate an XML digital signature and forward an unsigned message to CICS. In this case, it is still possible for the service requester's identity to flow with the unsigned message so that it can be used for CICS resource authorization checking.

Performance

Security is often at odds with performance, that is, the most secure technologies are often expensive to implement. Inevitably, the need to implement a solution that meets the security requirements must be balanced against the need to meet the solution's performance objectives.

After you have a clear understanding of the security requirements and security implementation options, consider the following CICS performance related questions:

- ▶ Will you use transport based security or SOAP message security?

SSL/TLS is a mature technology that has been optimized over a long period of time, and there are ways of optimizing performance such as persistent TCP/IP connections and SSL session ID reuse. These optimizations mean that expensive security functions, such as SSL handshaking, can be avoided for service requests following the initial handshake.

WS-Security support, in comparison, is completely stateless, and expensive security functions, such as XML digital signature validation, are repeated for each service request.

In practice, the most optimum solution is often to use a combination transport based and SOAP message based security, for example, transport based security for confidentiality and data integrity and SOAP message based security for transport of security tokens.

- ▶ Where is authentication done and how many times?

The cost of authentication in CICS is dependent on the security token used in the authentication. Simple security tokens like UsernameTokens are less expensive than binary security tokens such as X.509 certificates. More advanced authentication using a STS will incur an additional overhead.

If authentication is done by an intermediary server, where possible, the intermediary server should flow an asserted identity to CICS. This avoids the overhead of authenticating multiple times.

- ▶ Are you using hardware cryptographic devices in an optimal way?

Cryptographic hardware and ICSF (Integrated Cryptographic Hardware Facility) is a prerequisite for CICS XML digital signature and XML encryption processing. It is also required in order to maximize performance when CICS is configured to use SSL/TLS.

- ▶ What cipher suite are you using?

To utilize hardware cryptography, the chosen cipher suite algorithm must be available in hardware.

- ▶ Is there a requirement for an SOA appliance?

An SOA appliance such as WebSphere DataPower can be used in conjunction with CICS Web services to help secure the services and to offload expensive operations by processing the complex part of XML messages (such as an XML signature) at wirespeed.

WebSphere DataPower is also an ideal solution for other expensive tasks such as auditing and XML validation.

Archived

Elements of cryptography

When implementing security for a CICS Web services solution, it is useful to understand basic cryptography terminology and concepts such as:

- ▶ The difference between secret key cryptography and public key cryptography
- ▶ The definition of a hashing function, a digital signature, a digital certificate, a certificate authority, and a certificate revocation list

The main purpose of this chapter is to provide some background information about cryptography to help you when reading the following chapters in this book, in which we show security scenarios that demonstrate how cryptographic functions can be used with CICS.

Note: You do not need to be an expert in cryptography in order to implement a CICS Web services security solution. This chapter provides some detailed explanations of how cryptography works, and we realize that some of this information will be used for reference only.

3.1 The role of cryptography

As used in computer security, cryptography provides the following processes:

- ▶ *Encrypting* converts plaintext (that is, data in normal, readable form) into ciphertext, which conceals the meaning of the data to any unauthorized recipient. Encrypting is also called *enciphering*.

Most cryptographic systems combine two elements:

- An algorithm that specifies the mathematical steps needed to encrypt the data.
 - A cryptographic key (a string of numbers or characters), or keys. The algorithm uses the key to select one relationship between plaintext and ciphertext out of the many possible relationships the algorithm provides. The selected relationship determines the composition of the algorithm's result.
- ▶ *Decrypting* converts ciphertext back into plaintext. Decrypting is also called *deciphering*.
 - ▶ *Hashing* uses a one-way (irreversible) calculation to condense a long message into a compact bit string called a *message digest*.
 - ▶ Generating a *digital signature* involves encrypting a message digest with a private key to create the electronic equivalent of a handwritten signature. You can use a digital signature to verify the identity of the signer and to ensure that nothing has altered the signed document since it was signed.

In this chapter we show how you can use cryptography to achieve authentication, data integrity, confidentiality, and nonrepudiation.

3.2 Secret key (or symmetric) cryptography

In *secret key* cryptography, the sender and receiver of a message know and use the same secret key; the sender uses the secret key to encrypt the message, and the receiver uses the same secret key to decrypt the message. See Figure 3-1. Secret key cryptography is also known as symmetric cryptography.

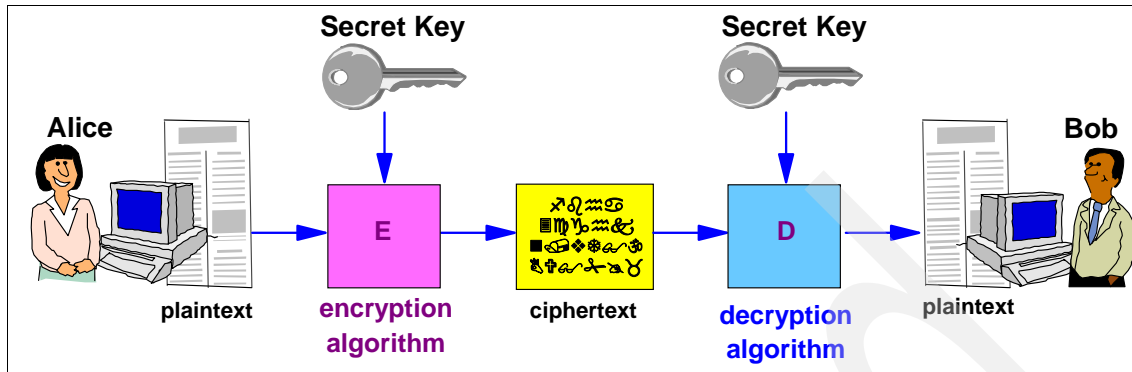


Figure 3-1 Secret key (or symmetric) cryptography

The main challenge of secret key cryptography is getting the sender and receiver to agree on the secret key without anyone else finding out. If the sender and receiver are in separate physical locations, they must trust a courier, a phone system, or some other transmission medium to prevent the disclosure of the secret key. Anyone who overhears or intercepts the key in transit can later read, modify, and forge all messages encrypted using that key.

Block ciphers

A *block cipher* is a type of secret key encryption algorithm that transforms a *fixed-length* block of plaintext data into a block of ciphertext data of the same length. This transformation takes place under the action of a user-provided secret key. Decryption is performed by applying the reverse transformation to the ciphertext block using the same secret key. The fixed length is called the block size. Common block sizes include 64 bits and 128 bits.

Iterated block ciphers

Iterated block ciphers encrypt a plaintext block by a process that has several rounds. In each round, the same transformation (also known as a round function) is applied to the data using a *subkey*. The set of subkeys is usually derived from the user-provided secret key by a special function. The set of subkeys is called the *key schedule*. The number of rounds in an iterated cipher depends on the desired security level and the consequent trade-off with performance. In most cases, an increased number of rounds will improve the security offered by a block cipher.

3.2.1 DES

The Data Encryption Algorithm (DEA), developed by IBM, is one example of an iterated block cipher. IBM submitted the DEA to the National Bureau of Standards (NBS) during an NBS public solicitation for cryptographic algorithms to be used in a Federal Information Processing Standard (FIPS). In 1977 the NBS issued FIPS Publication 46 *Data Encryption Standard (DES)*, which specified that the DEA be used within the United States Federal Government for the cryptographic protection of sensitive, but unclassified, computer data. As a result, the DEA is often called DES.

The DES was reaffirmed in 1983, 1988, 1993, and 1999. As time passed, the NBS became the National Institute of Standards and Technology or NIST; it is a division of the U. S. Department of Commerce.

The DES has a 64-bit block size. A DES key consists of 64 bits, of which 56 bits are randomly generated and used directly by the algorithm. The other 8 bits, which are not used by the algorithm, can be used for error detection. The binary format of the key is:

(B1,B2,...,B7,P1,B8,...,B14,P2,B15,...,B49,P7,B50,...,B56,P8)

Where (B1,B2,...,B56) are the independent bits of a DES key and (P1,P2,...,P8) are reserved for parity bits computed on the preceding seven independent bits and set so that the parity of the octet is odd, that is, there is an odd number of "1" bits in the octet.

DES modes of operation

When we use a block cipher to encrypt a message of *arbitrary* length, we use techniques known as modes of operation for the block cipher. In December, 1980, FIPS Publication 81 *DES Modes of Operation* announced four modes of operation for DES:

- ▶ Electronic Codebook (ECB)
- ▶ Cipher Block Chaining (CBC)
- ▶ Cipher Feedback (CFB)
- ▶ Output Feedback (OFB)

We now describe the first two of these modes of operation.

ECB

In ECB mode, the message M of arbitrary length is first divided into blocks m_i . Each block contains 64 bits, the block size of the DES algorithm. Each plaintext block m_i is used directly as the input block to the DES encryption algorithm with key k . The resultant output block is used directly as ciphertext. See Figure 3-2, where E_k represents encryption using the DES algorithm with k as the key.

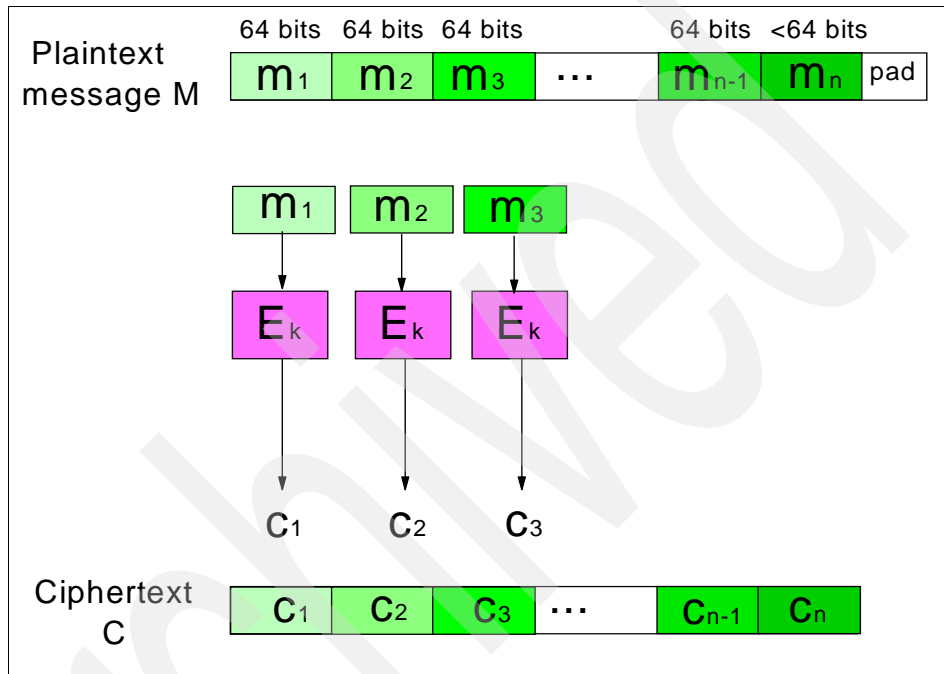


Figure 3-2 Electronic codebook (ECB) mode of operation

The analogy to a codebook arises because the same plaintext block always produces the same ciphertext block for a given cryptographic key. Thus a list (or codebook) of plaintext blocks and corresponding ciphertext blocks theoretically could be constructed for any given key.

Because the ECB mode is a 64-bit block cipher, an ECB device must encrypt data in integral multiples of 64 bits. If a user has less than 64 bits to encrypt, then the least significant bits of the unused portion of the input data block must be padded, for example, filled with random or pseudo-random bits prior to ECB encryption. The corresponding decrypting device must then discard these padding bits after decryption of the ciphertext block.

CBC

In practice, CBC is the most widely used mode of DES. In CBC, the message M of arbitrary length is first divided into blocks m_i . Each block contains 64 bits, the block size of the DES algorithm. Each plaintext block m_i is XORed (exclusive ORed) with the previous ciphertext block c_{i-1} and then encrypted. A 64-bit initialization vector c_0 is used as a “seed” for the process. See Figure 3-3, where a circle enclosing a cross represents an XOR operation.

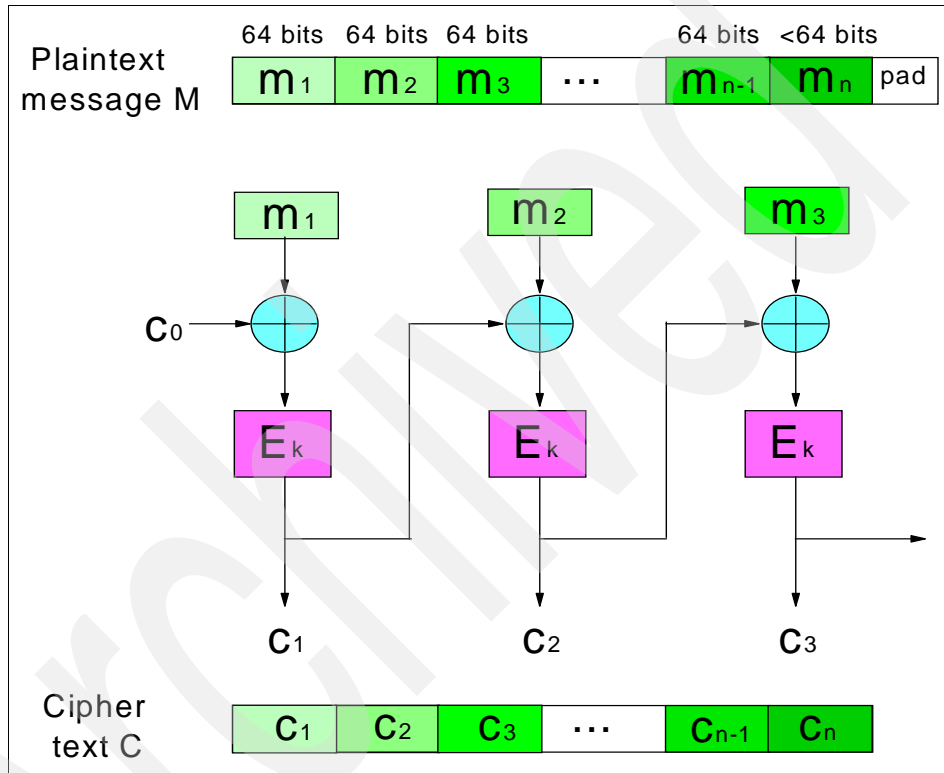


Figure 3-3 DES encryption using the Cipher Block Chaining (CBC) mode of operation

Thus, the encryption of each block depends on previous blocks, and the same 64-bit plaintext block can encrypt to different ciphertext blocks depending on its context in the overall message. XORing of the previous ciphertext block with the plaintext block conceals any patterns in the plaintext.

Partial data blocks (blocks of less than 64 bits) require special handling. One method of encrypting a final partial data block of a message is described next.

The following method can be used for applications where the length of the ciphertext can be greater than the length of the plaintext. In this case the final partial data block of a message is padded in the least significant bits positions with “0”s, “1”s, or pseudo-random bits. The decrypter will have to know when and to what extent padding has occurred. This can be accomplished explicitly, for example, using a padding indicator, or implicitly, for example, using constant length transactions.

The padding indicator will depend on the data being encrypted:

► Binary:

If the data is pure binary, then the partial data block should be left justified in the input block and the unused bits of the block set to the complement of the last data bit, that is, if the last data bit of the message is “0” then “1”s are used as padding bits and if the last data bit is “1” then “0”s are used. The input block is then encrypted.

The resulting output block is the ciphertext. The ciphertext message must be marked as being padded so that the decrypter can reverse the padding process, remove the padding bits and produce the original plaintext. The decrypter scans the decrypted padded block and discards the least significant bits that are all identical.

► Bytes:

If the data consists of bytes (for example, 8-bit ASCII characters) then the padding indicator should be a character denoting the number of padding bytes, including itself, and should be placed in the least significant byte of the input block before encrypting. For example, if there are five ASCII data characters in the final partial block of a message to be encrypted, then an ASCII “3” is put in the least significant byte of the input block (any pad characters can be used in the other two pad positions) before encryption. Again the ciphertext message must be marked as being padded.

Figure 3-4 shows the decryption of a message using the CBC mode of operation; D_k represents decryption using the DES algorithm with key k .

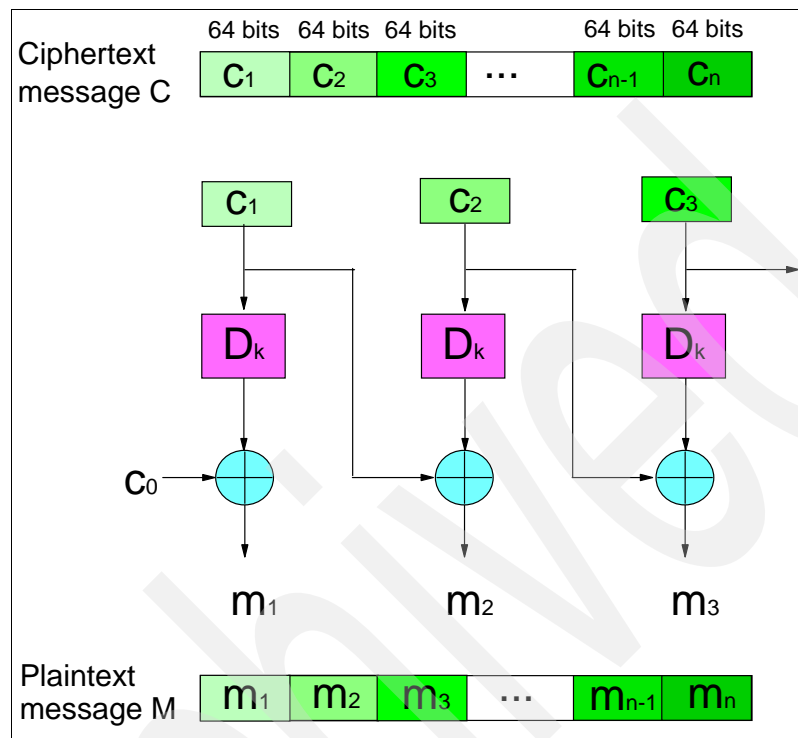


Figure 3-4 DES decryption using the CBC mode of operation

Status of DES

Because the speed of computers has increased significantly since 1977, it might now be possible to try every possible DES 56-bit key in turn until the correct key is identified. This technique of attempting to decipher a message is called exhaustive key search or brute force search. Indeed, a DES cracking machine has been used to recover a DES key in 22 hours.

Therefore, the consensus of the cryptographic community is that DES is no longer secure. FIPS 46-3 reaffirmed DES usage as of October 1999, but permitted single DES only for legacy systems. FIPS 46-3 included a definition of triple-DES (TDEA) which became “the FIPS approved symmetric encryption algorithm of choice.” On November 26, 2001, the NIST published FIPS 197 announcing the Advanced Encryption Standard (AES); the standard became effective on May 26, 2002. The NIST withdrew FIPS 46-3 on May 19, 2005.

3.2.2 Triple DES (TDEA)

For some time it has been common practice to protect information with triple-DES instead of DES. This means that the input data is, in effect, encrypted three times. There are a variety of ways of doing this; FIPS Pub 46-3 defines triple-DES encryption with keys k_1 , k_2 , and k_3 as:

$$C = E_{k_3}(D_{k_2}(E_{k_1}(M)))$$

Where $E_k(I)$ and $D_k(I)$ denote DES encryption and DES decryption, respectively, of the input I with the key k . See Figure 3-5.

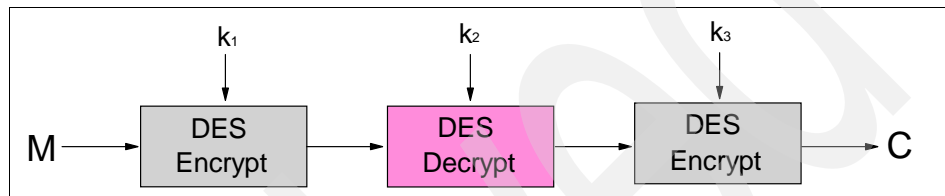


Figure 3-5 Triple DES - EDE

This mode of encryption is sometimes referred to as DES-EDE (encrypt, decrypt, encrypt). FIPS Pub 46-3 defines three keying options for DES-EDE:

- ▶ k_1 , k_2 and k_3 are independent.
Since each key has a length of 64 bits, this is sometimes referred to as TDEA-192. However, since only 56 bits of the 64 bits of each key are actually used, it is sometimes also called TDEA-168.
- ▶ k_1 and k_2 are independent, but $k_3 = k_1$.
This can be called TDEA-128 or TDEA-112.
- ▶ $k_1 = k_2 = k_3$.

Another variant is DES-EEE, which consists of three consecutive encryptions.

TDEA modes of operation

Like all block ciphers, triple-DES can be used in a variety of modes. The American National Standards Institute (ANSI) X9.52 standard *Triple Data Encryption Algorithm Modes of Operation* describes seven different modes:

- ▶ TDEA Electronic Codebook (TECB)
- ▶ TDEA Cipher Block Chaining (TCBC)
- ▶ TDEA Cipher Block Chaining - Interleaved (TCBC - I)
- ▶ TDEA Cipher Feedback (TCFB)

- ▶ TDEA Cipher Feedback - Pipelined (TCFB-P)
- ▶ TDEA Output Feedback (TOFB)
- ▶ TDEA Output Feedback - Interleaved (TOFB-I)

Figure 3-6 shows TDEA running in TECB mode.

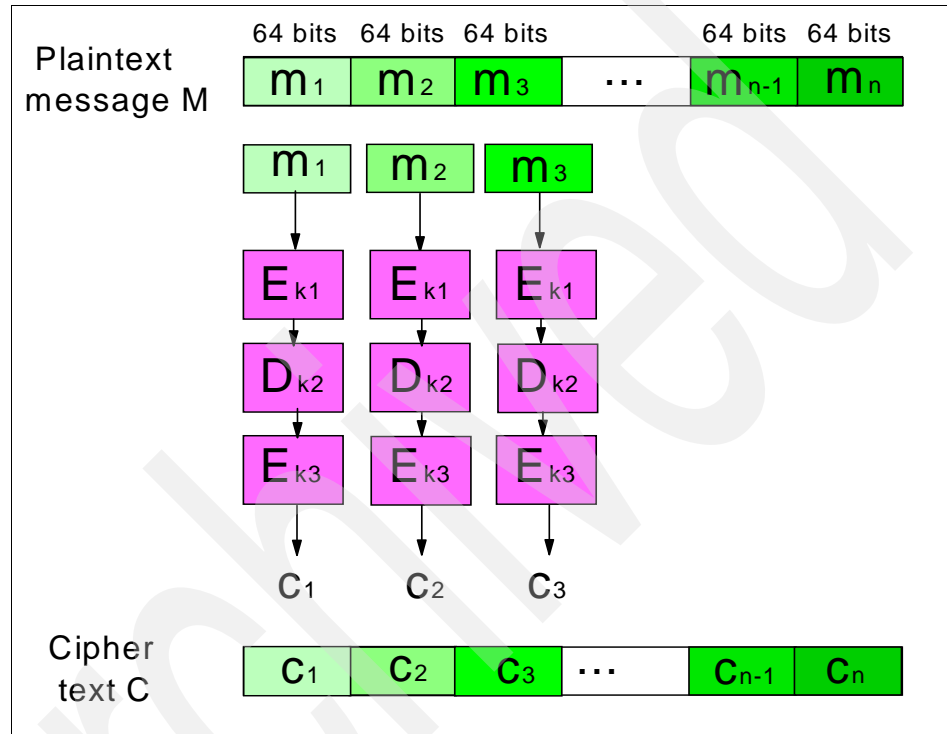


Figure 3-6 Triple DES running in TECB mode

3.2.3 AES

The Advanced Encryption Standard (AES) is another example of an iterated block cipher. The AES algorithm resulted from a multi-year evaluation process led by the NIST with submissions and review by an international community of cryptography experts. The Rijndael algorithm, invented by Joan Daemen and Vincent Rijmen, was selected as the standard. The NIST specified the AES in FIPS PUB 197 in November, 2001.

The AES processes data blocks of 128 bits. That is, the input and the output for the AES algorithm each consists of a sequence of 128 bits (16 bytes or 4 words).

The cipher key for the AES algorithm is a sequence of 128, 192, or 256 bits. These different “flavors” of AES can be referred to as “AES-128”, “AES-192”, and “AES-256”. The number of words N_k in the key is thus 4, 6, or 8.

The number of rounds N_r to be performed during the execution of the algorithm depends on the key size. When $N_k = 4$, then $N_r = 10$. If $N_k=6$, then $N_r=12$, and when $N_k=8$, then $N_r=14$.

3.3 Public key (or asymmetric) cryptography

In *public key* cryptography each person gets a pair of keys, one called the *public* key and the other called the *private* key. The public key is published, while the private key is kept secret. The need for the sender and receiver to share secret information is eliminated; all communications involve only public keys, and no private key is ever transmitted or shared. In this system, it is no longer necessary to trust the security of some means of communication. The only requirement is that public keys be associated with their users in a trusted manner (for instance, in a trusted directory).

In public key cryptography:

- ▶ Data encrypted with a public key can only be decrypted with the corresponding private key. This guarantees data privacy for the receiver, since he is the only one able to decrypt the data. But the receiver cannot be sure who the sender is; it could be anyone.
- ▶ Data encrypted with a private key can only be decrypted with the corresponding public key. Anyone can decrypt the data, but the receiver knows who the sender is because the data can come only from one sender, the owner of the private key.

When Alice wants to send a secret message to Bob, she looks up Bob’s public key in a directory, uses it to encrypt the message and sends it off. Bob then uses his private key to decrypt the message and read it. No one listening in can decrypt the message. Anyone can send an encrypted message to Bob, but only Bob can read it (because only Bob knows Bob’s private key). See Figure 3-7 on page 80. Public key cryptography is also known as asymmetric cryptography.

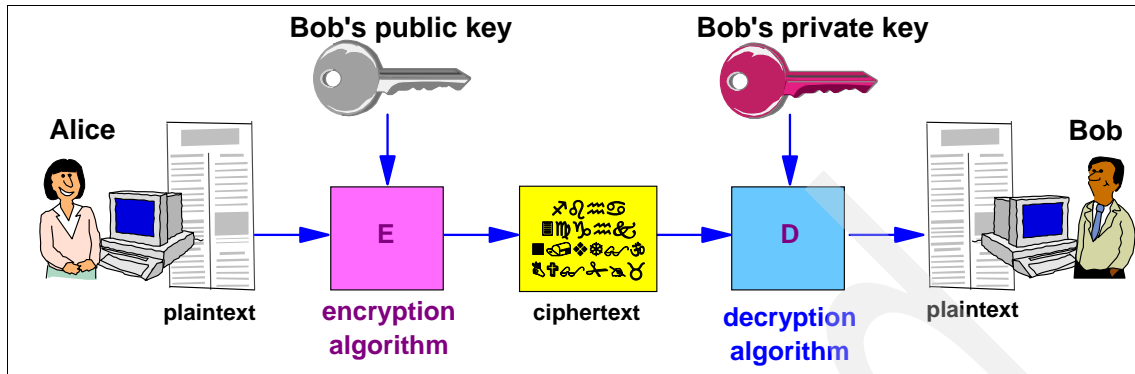


Figure 3-7 Public key (or asymmetric) cryptography

Note that public key cryptography solves the problem of how to safely transmit a secret key. When Alice wants to send a secret key to Bob, she looks up Bob's public key in a directory, uses it to encrypt the secret key and sends it off. Bob then uses his private key to decrypt the secret key and read it. No one listening in can decrypt the secret key.

In a public key cryptosystem, the private key is always linked mathematically to the public key. Therefore, it is always possible to attack a public key system by deriving the private key from the public key. Typically, the defense against this is to make the problem of deriving the private key from the public key as difficult as possible. For instance, some public key cryptosystems are designed such that deriving the private key from the public key requires the attacker to factor a large number; in this case it is computationally not feasible to perform the derivation.

3.3.1 RSA

The RSA cryptosystem is a public key cryptosystem developed in 1977 by Ronald Rivest, Adi Shamir, and Leonard Adleman. RSA stands for the first letter in each of its inventors' last names. The RSA algorithm is by far the most widely used public key cryptosystem in the world.

Note: Some information in this section was derived from the RSA Web site at www.rsasecurity.com/rsalabs/. The authors would like to thank RSA for permission to use this material.

Before we discuss how the RSA algorithm works, we review these definitions:

► Prime number

A prime number is any integer greater than 1 that is divisible only by 1 and itself. The first twelve primes are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, and 37.

► Factor

Given an integer n , any number that divides it is called a factor of n . For example, 7 is a factor of 91, because $91/7$ is an integer.

► Factoring

Factoring is the breaking down of an integer into its prime factors. For example, $140 = 2^2 \times 5 \times 7$. This is a hard problem (that is, a computationally intensive problem; one that is computationally difficult to solve).

► Relatively prime

Two integers are relatively prime if they have no common factors except 1. For example, 14 and 25 are relatively prime, while 14 and 91 are not (7 is a common factor of 14 and 91).

► Congruent modulo n

Given integers a , b , and n with $n > 0$, we say that a and b are congruent modulo n if $a-b$ is divisible by n , that is, if $(a-b)/n = i$, an integer. Equivalently, a and b are congruent modulo n if there is an integer i such that $a-b = i \times n$, that is, such that $a = b + (i \times n)$. If a and b are congruent modulo n , we write

$$a = b \pmod{n}$$

For example, $50 = 0 \pmod{5}$ because $(50 - 0)/5 = 10$, an integer. But 50 is not congruent to $1 \pmod{5}$ because $(50-1)/5$ is not an integer. However, 51 is congruent to $1 \pmod{5}$.

In arithmetic mod n , integers between 0 and $n - 1$ are used with normal addition, subtraction, multiplication, and exponentiation, except that after each operation the result keeps only the remainder after dividing by n . For example, $5^6 \pmod{23} = 8$ because $5^6 = 5 \times 5 \times 5 \times 5 \times 5 \times 5 = 15,625$ and 15,625 divided by 23 gives a quotient of 679 and a remainder of 8. Also, $3 + 4 = 2 \pmod{5}$ because $3 + 4 = 7$ and 7 divided by 5 gives a quotient of 1 and a remainder of 2.

The RSA algorithm works as follows: Take two large primes, p and q , and compute their product $n = pq$. Choose a number, e , less than n and relatively prime to $(p-1)(q-1)$. Find another number d such that $(ed - 1)$ is divisible by $(p-1)(q-1)$, that is, $ed = 1 \pmod{(p-1)(q-1)}$ or $d = e^{-1} \pmod{(p-1)(q-1)}$. The public key is the pair (n, e) ; the private key is (n, d) . For example, we take the two primes $p = 7$ and $q = 17$. Their product is $n = 119$. The number $e = 5$ is relatively prime to $(7-1)(17-1) = 96$. The number $d = 77$ is such that $ed-1 = 385-1 = 384$ is

divisible by 96. The public key is the pair (119, 5) and the private key is the pair (119,77). The factors p and q can be destroyed or kept with the private key.

RSA uses the following terminology:

- ▶ n is called the modulus
- ▶ e is called the public exponent
- ▶ d is called the private exponent

It is currently difficult to obtain the private exponent d from the public key (n, e). However, if one could factor n into p and q, then one could obtain the private exponent d. Thus the security of the RSA system is based on the assumption that factoring is difficult. The discovery of an easy method of factoring would “break” RSA.

Here is how the RSA system could be used for encryption. Suppose Alice wants to send a message m to Bob. Alice creates the ciphertext c by exponentiating: $c = m^e \text{ mod } n$, where e and n are Bob’s public key. She sends c to Bob. To decrypt, Bob also exponentiates: $m = c^d \text{ mod } n$; the relationship between e and d ensures that Bob correctly recovers m. Since only Bob knows d, only Bob can decrypt this message.

As an example, let us encrypt the message “sell” using the public key (119, 5). If we use the convention that a=1, b=2, c=3, and so forth to convert the letters to numbers, we find that the plaintext “sell” becomes 19 5 12 12:

- ▶ Raising 19 to the 5th power gives 2,476,099. Dividing 2,476,099 by $n=119$, we get a remainder of 66.
- ▶ Raising 5 to the 5th power gives 3125. Dividing 3125 by $n=119$, we get a remainder of 31.
- ▶ Raising 12 to the 5th power gives 248,832. Dividing 248,832 by $n=119$, we get a remainder of 3.

Thus our ciphertext is 66 31 3 3.

To decrypt 66 31 3 3, we use the private key (119, 77):

- ▶ Raising 66 to the 77th power gives 127316015002712725024996... (a very large number). Dividing that very large number by $n=119$, we get a remainder of 19.
- ▶ Raising 31 to the 77th power gives 683676142775442000196395... (another very large number). Dividing that number by $n=119$, we get a remainder of 5.
- ▶ Raising 3 to the 77th power gives 547440108942021938207715.... Dividing that number by $n=119$, we get a remainder of 12.

Thus our plaintext is 19 5 12 12 or “sell”.

3.3.2 Digital envelopes

In practice, the RSA system is often used together with a secret-key cryptosystem, such as DES. Suppose Alice wants to send an encrypted message to Bob. She first encrypts the message with DES, using a randomly chosen DES key. Then she looks up Bob's public key and uses it to encrypt the DES key. The DES-encrypted message and the RSA-encrypted DES key are sent to Bob. Upon receiving them, Bob decrypts the DES key with his private key, then uses the DES key to decrypt the message itself.

See Figure 3-8. This process is sometimes referred to as sending a digital envelope; it combines the high speed of DES with the key management convenience of the RSA system.

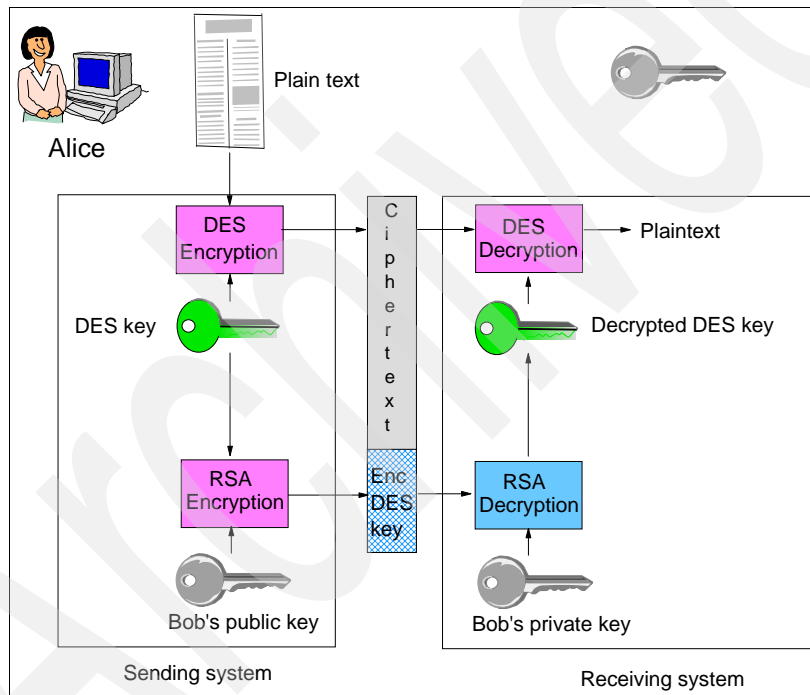


Figure 3-8 Using DES to encrypt data and RSA to manage the DES key

3.3.3 Appropriate key size in the RSA cryptosystem

The size of a key in the RSA algorithm typically refers to the size of n . The two primes, p and q , which compose n , should be of roughly equal length; this makes n harder to factor than if one of the primes is much smaller than the other. If one chooses to use a 768-bit value for n , the primes should each have a length of approximately 384 bits.

The best size for n depends on the user's security needs. The larger the value of n , the greater the security, but also the slower the RSA algorithm operations. Choose a length for n upon consideration, first, of the value of the protected data and how long it needs to be protected, and, second, of how powerful the potential threats might be.

Key sizes of 512-bits no longer provide sufficient security for anything more than very short-term security needs. RSA Laboratories currently recommends key sizes of 1024 bits for corporate use and 2048 bits for extremely valuable keys like the root key pair used by a certifying authority. Less valuable information might well be encrypted using a 768-bit key, since such a key is still beyond the reach of all known key breaking algorithms. RSA Laboratories publishes recommended key lengths on a regular basis.

As for the slowdown caused by increasing the key size, doubling the length of n will, on average, increase the time required for public key operations (encryption and signature verification) by a factor of four, and increase the time taken by private key operations (decrypting and signing) by a factor of eight. Key generation time would increase by a factor of 16 upon doubling the length of n , but this is a relatively infrequent operation for most users.

3.4 Hash functions

A hash function H is a transformation that takes an input message m and returns a fixed-size string, which is called the hash value h . Using mathematical notation for functions, we express this as $h=H(m)$. See Figure 3-9.

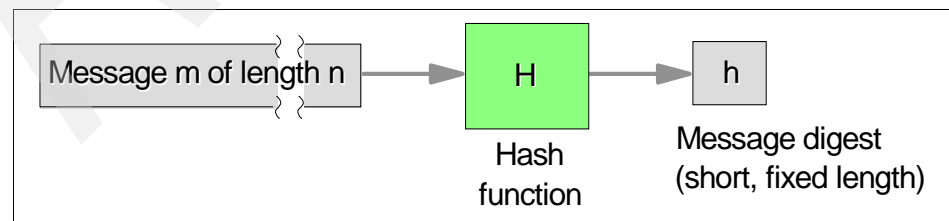


Figure 3-9 A hash function

Figure 3-10 shows an example of a hash function at work.

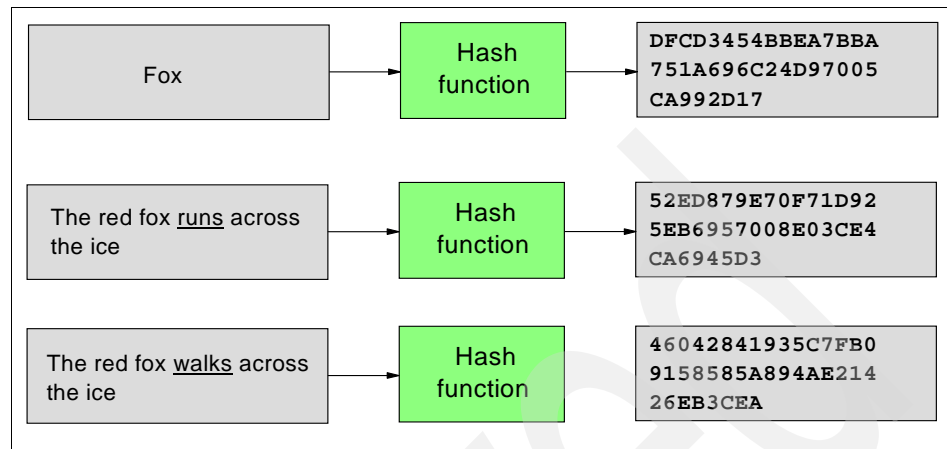


Figure 3-10 A hash function at work

Notice that the hash function in Figure 3-10 produces a message digest whose length is 20 bytes regardless of the length of the message. Notice also that if we make a small change in the message we get a very different message digest.

When employed in cryptography, hash functions are usually chosen to have some additional properties. The basic requirements for a cryptographic hash function are as follows:

- ▶ The input can be of any length.
- ▶ The output has a fixed length.
- ▶ $H(m)$ is relatively easy to compute for any given m .
- ▶ $H(m)$ is one-way.

A hash function H is said to be one-way if it is hard to invert, where “hard to invert” means that given a hash value h , it is computationally not feasible to find some input m such that $H(m)=h$. An everyday example of a one-way function is mashing a potato; you can do it easily, but once you have mashed the potato, you will find it rather difficult to reconstruct the original potato.

- ▶ $H(m)$ is collision-free.

A collision-free hash function H is one for which it is computationally not feasible to find any two messages x and y such that $H(x)=h$ and $H(y)=h$; that is, it is computationally not feasible to find any two messages that hash to the same value.

The hash value represents concisely the longer message or document from which it was computed; this value is called the *message digest*. One can think of a message digest as a *digital fingerprint* of the larger document; it identifies the message much like a real fingerprint identifies a person. Thus a good cryptographic hash function ensures that it is very difficult to:

- ▶ Recover the message from the message digest
- ▶ Construct a block of data M_2 that has the same message digest h as another given block M_1

You can use a hashing function to verify that data has not been altered during transmission. The sender of the data calculates the message digest using the data itself and the hashing function. The sender then ensures that the message digest is transmitted *with integrity* to the intended receiver of the data; one way to do this is to publish the message digest in a reliable source of public information. When the receiver gets the data, he can generate the message digest and compare it to the original one. If the two are equal, he can accept the data as genuine; if they differ, he can assume that the data is bogus. See Figure 3-11.

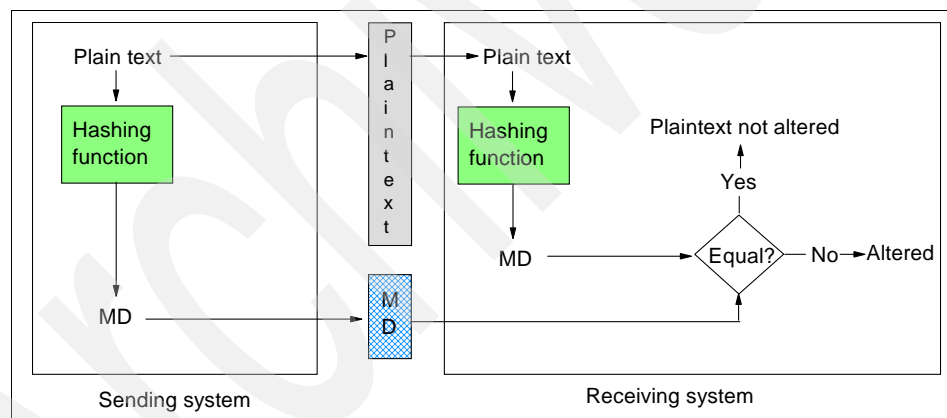


Figure 3-11 Using a hash function to verify that data has not been altered

In the preceding example, the message digest should not be sent in the clear. Since the hash functions are well-known and no key is involved, a man-in-the-middle could not only forge the message but also replace the message digest with that of the forged message. This would make it impossible for the receiver to detect the forgery.

I. Damgard and R.C. Merkle greatly influenced cryptographic hash function design by defining a hash function in terms of what is called a *compression function*. A compression function takes a fixed-length input (for example, 512 bits) and returns a shorter, fixed-length output (for example, 160 bits).

Given a compression function F , a hash function can be defined by repeated applications of the compression function F until the entire message has been processed. In this process, a message of arbitrary length is broken into blocks whose length depends on the compression function, and *padded* (for security reasons) so the size of the message is a multiple of the block size. The blocks are then processed sequentially, taking as input the result of the hash so far and the current message block, with the final output being the hash value for the message. See Figure 3-12.

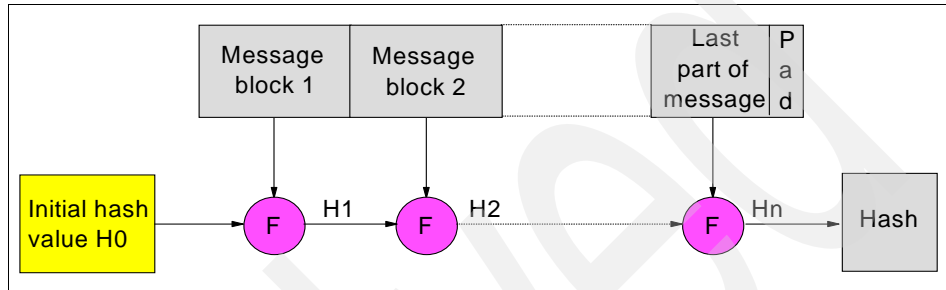


Figure 3-12 Iterative structure for hash functions

Here are some of the well-known hash functions:

- ▶ MD2 and MD5

MD2 and MD5 were developed by Ronald Rivest of the Laboratory for Computer Science at the Massachusetts Institute of Technology (MIT). Both functions take a message of arbitrary length and produce a 128-bit message digest. MD2 was optimized for 8-bit machines, whereas MD5 was aimed at 32-bit machines. Description and source code for MD2 and MD5 can be found as Internet RFCs 1319 and 1321, respectively.

- ▶ SHA-1

The Secure Hash Algorithm (SHA) was developed by the NIST and specified in the Secure Hash Standard (FIPS PUB 180). SHA-1 corrected an unpublished flaw in SHA and was published in 1994 as FIPS PUB 180-1.

SHA-1 is an iterative hash function, as shown in Figure 3-12. It operates on messages whose length is less than 2^{64} bits. The message is first padded so that the length in bits of the message is a multiple of 512. Then the message is parsed into 512-bit message blocks.

The algorithm is slightly slower than MD5 but the larger message digest makes it more secure against brute-force collision and inversion attacks.

- ▶ SHA-256

FIPS PUB 180-2 specifies the SHA-256 algorithm. This algorithm also takes a message of less than 2^{64} bits in length but it produces a 256-bit message digest.

3.5 Message authentication codes

A message authentication code (MAC) is a short piece of information used to authenticate a message. It is an authentication tag derived by applying an authentication scheme, together with a secret key, to a message. Unlike digital signatures, MACs are computed and verified with the *same* key, so that they can only be verified by the intended recipient. The MAC value protects both a message's integrity and its authenticity.

There are four types of MACs:

- ▶ Unconditionally secure
- ▶ Stream cipher-based
- ▶ Block cipher-based
- ▶ Hash function-based

We briefly discuss block cipher-based MACs and hash function-based MACs.

3.5.1 Block cipher-based MACs

Figure 3-13 shows how we might use the DES algorithm to compute a MAC on a message M . We begin by dividing the message M into blocks m_i . Each block contains 64 bits, the block size of the DES algorithm. We XOR each block m_i with the previous ciphertext block c_{i-1} and then encrypt it by using the DES encryption algorithm with key k . A 64-bit initialization vector c_0 is used as a “seed” for the process. The output from the last step is the MAC.

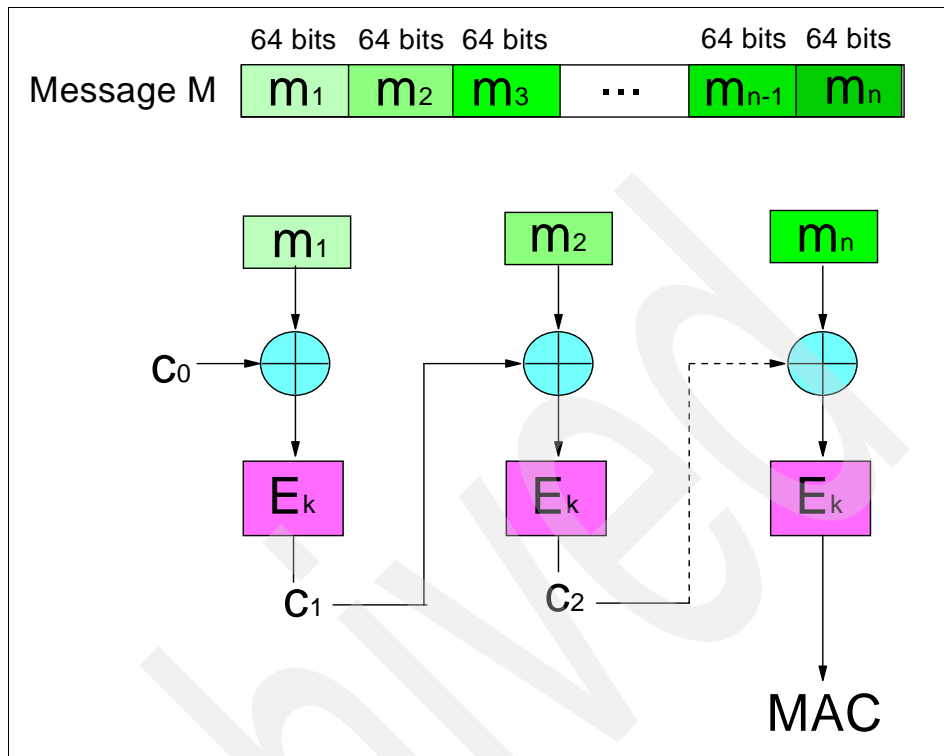


Figure 3-13 Computing a MAC by using the DES block cipher

3.5.2 Hash function-based MACs

MACs based on cryptographic hash functions are known as HMACs. HMACs have two functionally distinct parameters: a *message input*, and a *secret key* known only to the message originator and intended receivers.

An HMAC function is used by the message sender to produce a value (the MAC) that is formed by condensing the secret key and the message input. The MAC is typically sent to the message receiver along with the message. The receiver computes the MAC on the received message using the same key and HMAC function as was used by the sender, and compares the result computed with the received MAC. If the two values match, the message has been correctly received, and the receiver is assured that the sender is a member of the community of users that share the key. See Figure 3-14.

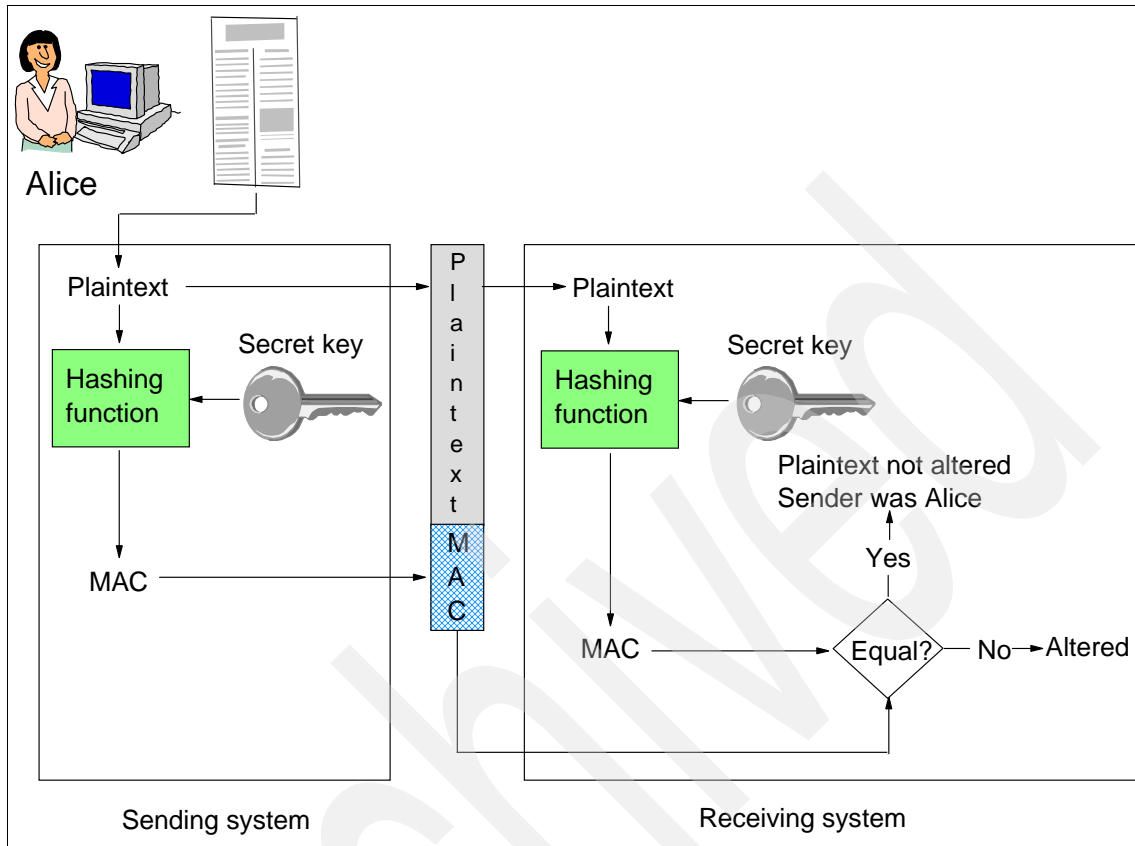


Figure 3-14 Keyed-hash message authentication code (HMAC)

Note that, since the receiver has the key that is used in creation of the MAC, this process does not offer a guarantee of nonrepudiation because it is theoretically possible for the receiver to forge a message and claim that it was sent by the sender.

3.6 Digital signatures

In this book *digital signature* is used to mean a cryptographically-based signature assurance scheme. We discuss two such schemes: the Digital Signature Algorithm (DSA) and RSA. While the DSA can only be used to provide digital signatures, the RSA system can be used for both encryption and digital signatures.

3.6.1 Using DSA for digital signatures

The NIST published the first version of the DSA in the *Digital Signature Standard (DSS)* FIPS PUB 186 in May, 1994. The current version was published in FIPS PUB 186-2 in January, 2000; in October, 2001, Change Notice 1 amended FIPS PUB 186-2.

DSA parameters

The DSA makes use of the following parameters:

- ▶ p = a prime number where $2^{1023} < p < 2^{1024}$
- ▶ q = a prime divisor of $p-1$, where $2^{159} < q < 2^{160}$
- ▶ $g = h^{(p-1)/q} \bmod p$, where h is any integer with $1 < h < p-1$ such that $h^{(p-1)/q} \bmod p > 1$
- ▶ x = a randomly or pseudo randomly generated integer with $0 < x < q$
- ▶ $y = g^x \bmod p$
- ▶ k = a randomly or pseudo randomly generated integer with $0 < k < q$

Appendix 2 and Appendix 3 of FIPS PUB 186-2 specify methods for generating p , q , x , and k .

The integers p , q , and g can be public and can be common to a group of users. The integers x and y are a user's private and public keys, respectively. Parameters x and k are used for signature generation only, and must be kept secret. Parameter k must be regenerated for each signature.

DSA signature generation

The signature of a message M is the pair of numbers r and s computed according to the equations:

$$r = (g^k \bmod p) \bmod q$$

$$s = (k^{-1}(\text{SHA-1}(M) + xr)) \bmod q$$

In these equations, k^{-1} is the multiplicative inverse of k , mod q ; that is, $(k^{-1} k) \bmod q = 1$ and $0 < k^{-1} < q$. The value of $\text{SHA-1}(M)$ is a 160-bit string output by the Secure Hash Algorithm SHA-1. For use in computing s , this string must be converted to an integer.

DSA signature verification

Prior to verifying the signature in a signed message, p , q , and g plus the sender's public key y and identity are made available to the verifier in an authenticated manner.

Let M' , r' , and s' be the received versions of M , r , and s respectively. See Figure 3-15.

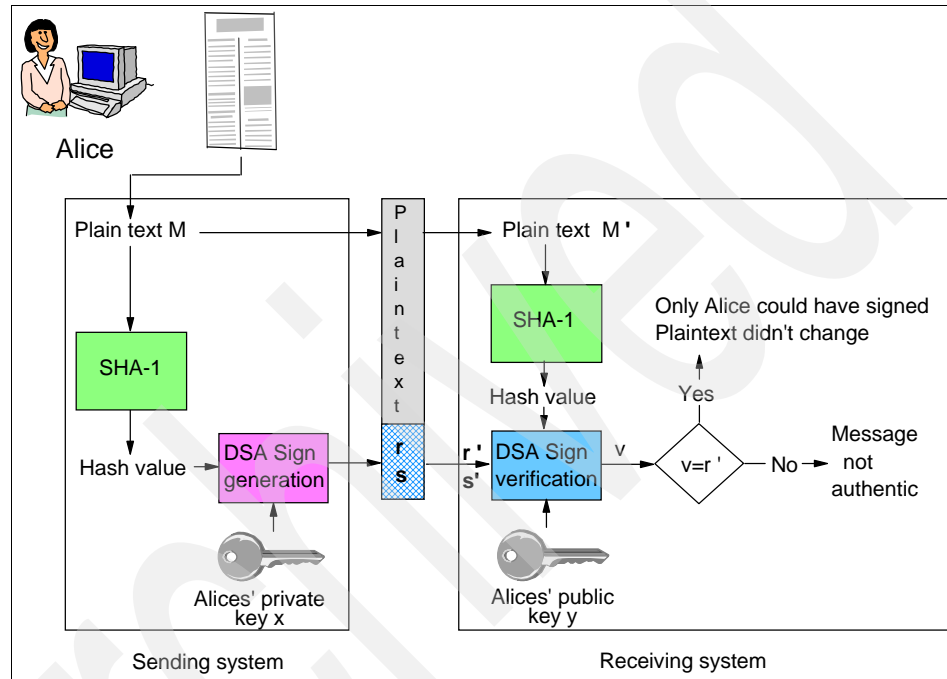


Figure 3-15 Verifying a DSA signature

To verify the signature, the verifier first checks to see that $0 < r' < q$ and $0 < s' < q$; if either condition is violated the signature should be rejected. If these two conditions are satisfied, the verifier computes:

$$w = (s')^{-1} \bmod q$$

$$u1 = ((\text{SHA-1}(M'))w) \bmod q$$

$$u2 = (r')w \bmod q$$

$$v = ((g^{u1} y^{u2}) \bmod p) \bmod q$$

If $v = r'$, then the signature is verified and the verifier can have high confidence that the received message was sent by the party holding the secret key x corresponding to y .

If v does not equal r' , then the message might have been modified, the message might have been incorrectly signed by the signatory, or the message might have been signed by an impostor. The message should be considered invalid.

The ANSI X9.62 standard *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)* specifies a method of providing digital signatures that makes use of the properties of mathematical objects known as elliptic curves. Since CICS TS V3.1 does not support this method, we do not discuss it here.

3.6.2 Using RSA for digital signatures

When RSA cryptography is used to calculate a digital signature, the sender encrypts the message digest of the document with his or her own private key. Anyone with access to the public key of the signer can verify the signature.

Suppose that Alice wants to send a signed document or message to Bob. She applies a hash function to the message, creating a message digest. She then encrypts the message digest with her private key, thereby creating the digital signature. (Because the message digest is usually considerably shorter than the original message, Alice saves a considerable amount of time when she encrypts the message digest rather than the message itself).

Alice sends Bob the encrypted message digest (digital signature) and the message. Bob, upon receiving the message and signature, decrypts the signature with Alice's public key to recover the message digest. He then hashes the message with the same hash function Alice used and compares the result to the message digest decrypted from the signature.

If they are exactly equal, the signature has been successfully verified and he can be confident the message did indeed come from Alice. If they are not equal, then the message either originated elsewhere or was altered after it was signed, and he rejects the message. See Figure 3-16.

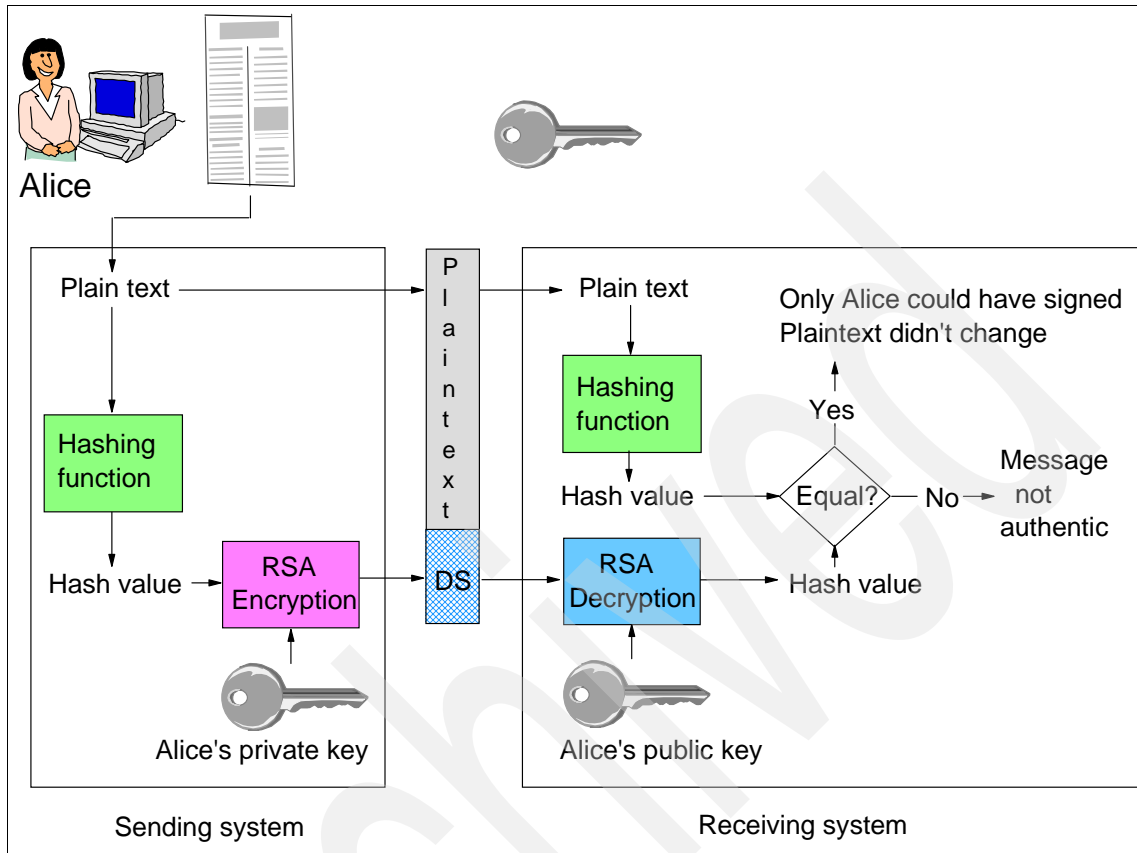


Figure 3-16 Creating and verifying a digital signature using RSA

In mathematical terminology, when Alice wants to send a message m to Bob, she applies a hash function H to the message m , creating a message digest $h=H(m)$. She then creates a digital signature s by exponentiating: $s = h^d \bmod n$, where d and n are Alice's private key. She sends m and s to Bob. To verify the signature, Bob exponentiates and checks that the message digest h is recovered: $h = s^e \bmod n$, where e and n are Alice's public key.

In practice, the public exponent in the RSA algorithm is usually much smaller than the private exponent. This means that verification of a signature is faster than signing. This is desirable when a message will be signed by an individual only once, but the signature might be verified many times.

Note that the recipient of signed data can use a digital signature to prove to a third party that the signature was in fact generated by the signatory. This is known as nonrepudiation because the signatory cannot, at a later time, repudiate the signature.

There is a potential problem with this type of digital signature. Alice not only signed the message she intended to sign, but she also signed all other messages that happen to hash to the same message digest. When two messages hash to the same message digest it is called a collision; the collision-free properties of hash functions are a necessary security requirement for most digital signature schemes. A hash function is secure if it is very time consuming, if at all possible, to figure out the original message given its digest.

In addition, someone could pretend to be Alice and sign documents with a key pair he claims is Alice's. To avoid scenarios such as this, there are digital documents called *certificates* that associate a person with a specific public key. We discuss digital certificates in 3.7, "Public key digital certificates" on page 97.

Suppose that Alice wants to keep the contents of the document secret instead of sending the document in the clear as in Figure 3-16 on page 94. In this case she might want to sign the document, then encrypt it using Bob's public key. Bob will then need to decrypt the document using his private key and verify the signature on the recovered message using Alice's public key. See Figure 3-17.

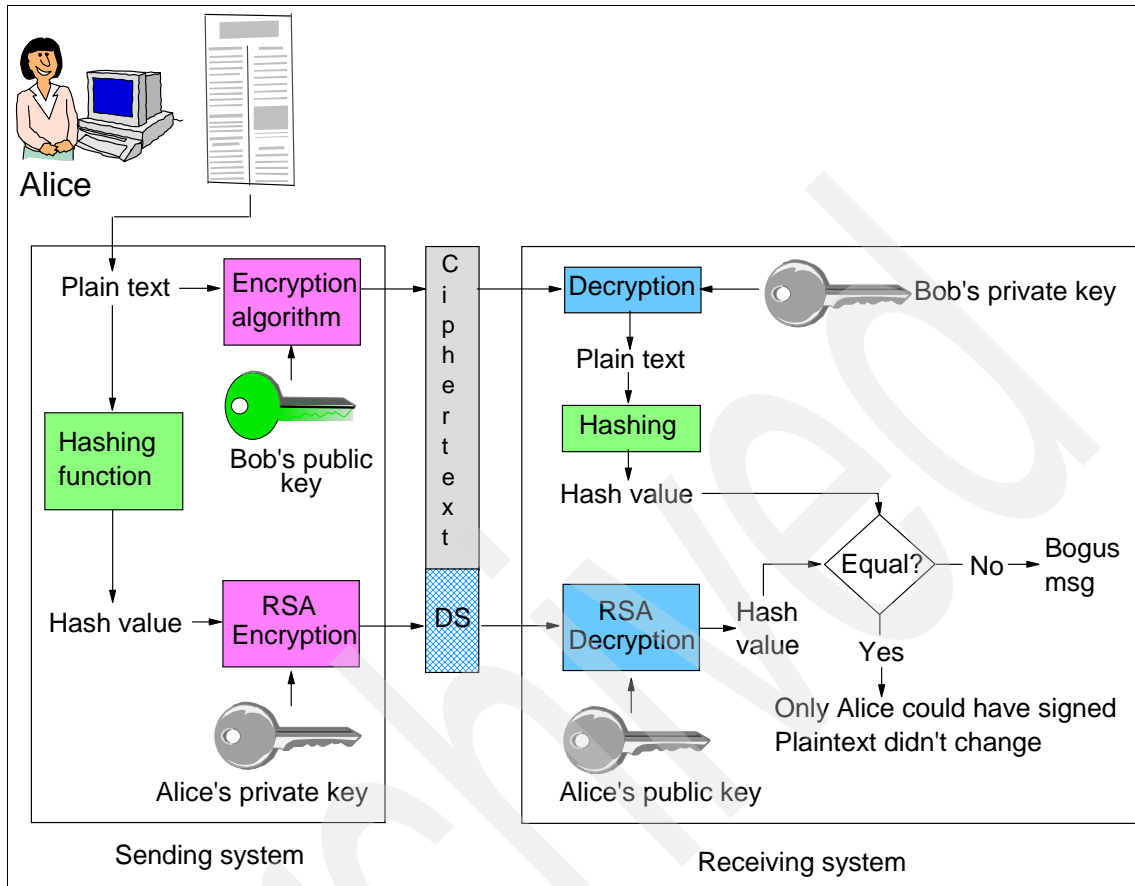


Figure 3-17 Creating and verifying a digital signature while encrypting the message

Alternatively, if it is necessary for intermediary third parties to validate the integrity of the message without being able to decrypt its content, a message digest can be computed on the encrypted message, rather than on its plaintext form.

3.6.3 Comparing RSA with DSA for digital signatures

In DSA, signature generation is faster than signature verification, whereas with the RSA algorithm, signature verification is very much faster than signature generation (if the public and private exponents, respectively, are chosen for this property, which is the usual case). It might be claimed that it is advantageous for signing to be the faster operation, but since in many applications a piece of digital information is signed once, but verified often, it might well be more advantageous to have faster verification.

3.7 Public key digital certificates

The tricky aspect of digital signatures is the trustworthy distribution of public keys, since the receiver requires a genuine copy of the sender's public key. This is provided by public key digital certificates.

A digital certificate is analogous to a passport in the following ways:

- ▶ Passports are issued by a trusted authority such as a government passport office. Digital certificates are issued by trusted authorities known as Certificate Authorities or CAs.
- ▶ A government passport office does not issue a passport unless the person who requests it has proven his identity and citizenship to the passport office. CAs have a responsibility to check the credentials provided in an application for a digital certificate. The CA might, for example, require the person who is requesting the certificate to appear in person and show a birth certificate.
- ▶ A passport certifies the bearer's name, address, and citizenship. A digital certificate establishes the subject's distinguished name (DN) and public key.
- ▶ Specialized equipment is used in the creation of a passport to make it very difficult to alter the information in it or to forge a passport altogether. CAs sign the digital certificates they issue with their private key.
- ▶ If other authorities, such as the border police in other countries, trust the authority that issued the passport, they implicitly trust the passport. If a Web user trusts a CA, he implicitly trusts digital certificates issued by the CA.
- ▶ Both passports and digital certificates are valid for a limited time.

Certificate issuance proceeds as follows. The requester generates a public and private key pair and then sends the public key to an appropriate CA with some proof of identification. The CA checks the identification and takes any other necessary steps to assure itself that the request really did come from the requester and that the public key was not modified in transit. Then the CA sends the requester a certificate which attests that the public key belongs to the requester.

In the following discussion of digital certificates, we use the description of the version 3 format of a digital certificate as given in RFC 3280 *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*.

The certificate is a sequence of three required fields:

▶ `tbsCertificate`

The `tbsCertificate` field contains the name of the subject of the certificate, a public key associated with the subject, the name of the issuer of the certificate, a validity period, and other associated information which we describe in 3.7.1, “`tbsCertificate`” on page 99.

▶ `signatureAlgorithm`

The `signatureAlgorithm` field contains the identifier for the cryptographic algorithm used by the CA to sign this certificate. RFC 3279 *Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile* lists the supported algorithms:

– `md2WithRSAEncryption`

This algorithm uses md2 for the hash function and RSA for the encryption algorithm.

– `md5WithRSAEncryption`

– `sha-1WithRSAEncryption`

– `id-dsa-with-sha1`

This algorithm uses SHA-1 for the hash function, and it uses the Digital Signature Algorithm.

– `ecdsa-with-SHA1`

▶ `signatureValue`

The `signatureValue` field contains a digital signature computed upon the `tbsCertificate`. The ASN.1 DER encoded `tbsCertificate` is used as the input to the signature function. This signature value is encoded as a BIT STRING and included in the signature field as shown in Figure 3-18.

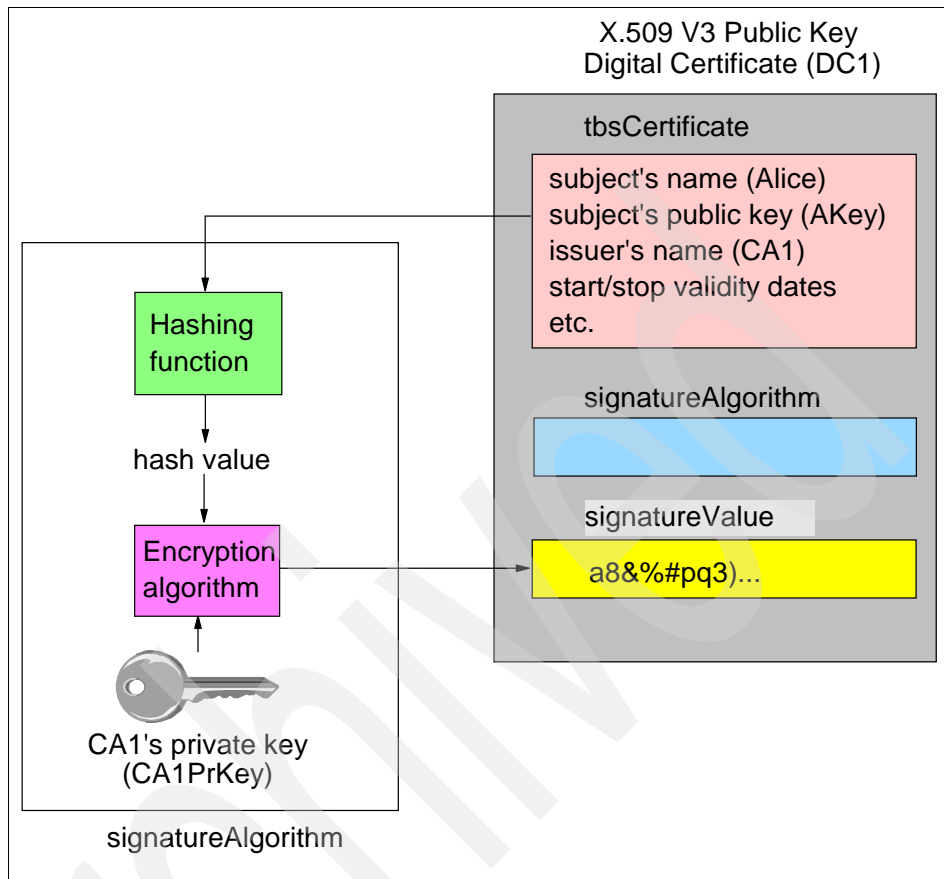


Figure 3-18 X.509 V3 public key digital certificate

By generating this signature, a CA certifies the validity of the information in the `tbsCertificate` field. In particular, the CA certifies the binding between the public key material and the subject of the certificate.

3.7.1 tbsCertificate

A `tbsCertificate` contains the following fields:

- ▶ `version`

ITU-T X.509, which was first published in 1988 as part of the X.500 Directory recommendations, defines a standard certificate format. (ITU-T is the International Telecommunications Union; it was formerly known as CCITT and is a multinational union that provides standards for telecommunications equipment and systems.)

The certificate format in the 1988 standard is called the version 1 format. When X.500 was revised in 1993, two more fields were added, resulting in the version 2 format. Experience gained in attempts to deploy RFC 1422 *Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management* revealed the need to develop a third version. In June, 1996, standardization of the basic version 3 format was completed. The value stored in the `version` field is one less than the version number; for example, when the version is 3, the value stored in the `version` field is 2.

- ▶ `serialNumber`

The serial number is a positive integer assigned by the CA to the certificate. It is unique for each certificate issued by a CA; that is, the issuer name and serial number identify a unique certificate.

- ▶ `signature`

This field contains the algorithm identifier for the algorithm used by the CA to sign the certificate. This field must contain the same algorithm identifier as the `signatureAlgorithm` field.

- ▶ `issuer`

The `issuer` field identifies the entity that has signed and issued the certificate. It contains a distinguished name (DN). We explain what a distinguished name is in “Distinguished names” on page 102.

- ▶ `validity`

The certificate validity period is the time interval during which the CA warrants that it will maintain information about the status of the certificate. The field is represented as a sequence of two dates:

- `notBefore` - The date on which the certificate validity period begins.
- `notAfter` - The date on which the certificate validity period ends.

Both `notBefore` and `notAfter` can be encoded `YYMMDDHHMMSSZ` or `YYYYMMDDHHMMSSZ`.

- ▶ `subject`

The `subject` field identifies the entity associated with the public key stored in the `subjectPublicKeyInfo` field. The `subject` field contains a DN.

If the subject is a CA, then the `subject` field must contain a DN that matches the contents of the `issuer` field in all certificates issued by the subject CA.

- ▶ `subjectPublicKeyInfo`

This field is used to carry the public key and identify the algorithm with which the key is used. RFC 3279 lists the following supported algorithm identifiers:

- rsaEncryption

When the algorithmIdentifier is rsaEncryption, the public key must be encoded as a sequence of two integers: the modulus n and the public exponent e .

- id-dsa

When the algorithmIdentifier is id-dsa, the public key must be encoded as the integer y .

- dhpublicnumber

This identifies the Diffie-Hellman key exchange algorithm. The public key is the integer $y = g^x \text{ mod } p$. We discuss the Diffie-Hellman key exchange algorithm in 3.9.2, “The Diffie-Hellman key agreement protocol” on page 114.

- id-keyExchangeAlgorithm

This identifies the Key Exchange Algorithm (KEA), which is a key agreement algorithm. We do not discuss it further.

- id-ecPublicKey

When the algorithmIdentifier is id-ecPublicKey, the public key is intended for use in either the Elliptic Curve Digital Signature Algorithm (ECDSA) or the Elliptic Curve Diffie-Hellman (ECDH) key exchange algorithm, neither of which is discussed further.

- ▶ issuerUniqueId (optional)

This field is used to handle the possibility of reuse of issuer names over time. RFC 3280 recommends that names not be reused for different entities and that Internet certificates not make use of unique identifiers.

- ▶ subjectUniqueId (optional)

This field is used to handle the possibility of reuse of subject names over time. RFC 3280 also recommends against the use of this field.

- ▶ extensions (optional)

If present, this field is a sequence of one or more certificate extensions. The extensions defined for X.509 V3 certificates provide methods for associating additional attributes with users or public keys and for managing a certification hierarchy. We discuss a few of the standard extensions defined in RFC 3280 in 3.7.2, “Standard extensions for X.509 V3 digital certificates” on page 104.

Example 3-1 shows the decode of an X.509 certificate as found at:

<http://en.wikipedia.org/wiki/X.509>

Certificate:

Data:

Version: 1 (0x0)
Serial Number: 7829 (0x1e95)
Signature Algorithm: md5WithRSAEncryption
Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc,
OU=Certification Services Division,
CN=Thawte Server CA/Email=server-certs@thawte.com
Validity
Not Before: Jul 9 16:04:02 1998 GMT
Not After : Jul 9 16:04:02 1999 GMT
Subject: C=US, SP=Maryland, L=Pasadena, O=Brent Baccala,
OU=FreeSoft, CN=www.freesoft.org/Email=baccala@freesoft.org
Subject Public Key Info:
Public Key Algorithm: rsaEncryption
RSA Public Key: (1024 bit)
Modulus (1024 bit):
00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8:
e8:35:1c:9e:27:52:7e:41:8f:
Exponent: 65537 (0x10001)
Signature Algorithm: md5WithRSAEncryption
93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:
ab:2f:4b:cf:0a:13:90:ee:2c:0e:43:03:be:f6:ea:8e:9c:67:
d0:a2:40:03:f7:ef:6a:15:09:79:a9:46:ed:b7:16:1b:41:72:
0d:19:aa:ad:dd:9a:df:ab:97:50:65:f5:5e:85:a6:ef:19:d1:
5a:de:9d:ea:63:cd:cb:cc:6d:5d:01:85:b5:6d:c8:f3:d9:f7:
8f:0e:fc:ba:1f:34:e9:96:6e:6c:cf:f2:ef:9b:bf:de:b5:22:
68:9f

Distinguished names

Both the issuer field and the subject field of tbsCertificate must contain an X.520 Distinguished Name (DN). A DN is a sequence of Relative Distinguished Names (RDNs). An RDN™ has the form <attribute type> = <value>. Table 3-1 on page 103 shows the string representations of common attribute types.

Table 3-1 Attribute types and their string representations

Attribute Type	String
countryName	C
organizationName	O
organizationalUnitName	OU
stateOrProvinceName	SP
localityName	L
commonName	CN

You can think of a DN as a unique name that unambiguously identifies a single entry in a directory information tree. Each RDN in a DN corresponds to a branch in the tree leading from the root of the tree to the directory entry.

As shown in Figure 3-19, the distinguished name C=US, O=IBM, OU=IO, SP=NY, L=End, CN=Bob Herman describes Bob Herman, who works in the village of Endicott in the state of New York, USA, for the Integrated Operations unit of IBM; while the distinguished name C=FR, O=IBM, OU=S&D, SP=Her, L=MOP, CN=Nigel Williams describes Nigel Williams, who works in the city of Montpellier in the province of Herault, France, for the Sales and Distribution unit of IBM.

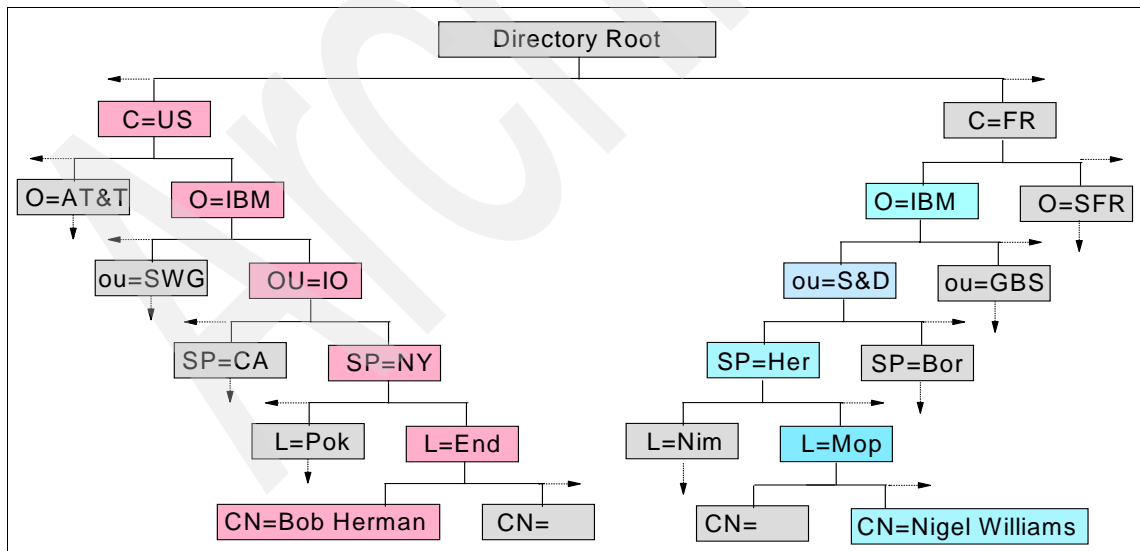


Figure 3-19 Distinguished names

3.7.2 Standard extensions for X.509 V3 digital certificates

RFC3280 defines sixteen standard extensions, but we limit our discussion here to only the most relevant ones; specifically, those supported by the RACF RACDERT command that you can use to generate a digital certificate. These extensions are:

► Key usage

The key usage extension defines the purpose of the subject public key contained in the certificate:

– digitalSignature (0)

The key is used with a digital signature mechanism to support security services other than certificate signing (bit 5) or certificate revocation list (CRL) signing (bit 6). Digital signature mechanisms are often used for entity authentication and data origin authentication with integrity. (We explain what a CRL is in 3.8, “Certificate revocation lists” on page 109.)

– nonRepudiation (1)

The key is used to verify digital signatures used to provide a nonrepudiation service, which protects against the signing entity falsely denying some action, excluding certificate or CRL signing. In case of later conflict, a reliable third party can determine the authenticity of the signed data.

– keyEncipherment (2)

The key is used for key transport. For example, when an RSA key is to be used for key management, then this bit is set.

– dataEncipherment (3)

The key is used for enciphering user data, other than cryptographic keys.

– keyAgreement (4)

The key is used for key agreement. For example, when a Diffie-Hellman key is to be used for key management, then this bit is set.

– keyCertSign (5)

The key is used for verifying a signature on public key certificates.

– cRLSign (6)

The key is used for verifying a signature on a CRL.

– encipherOnly (7)

The meaning of the encipherOnly bit is undefined in the absence of the keyAgreement bit. When the encipherOnly bit is asserted and the keyAgreement bit is also set, the key can be used only for enciphering data while performing key agreement.

- decipherOnly (8)

The meaning of this bit is the same as the encipherOnly bit except that it applies to a decipher operation.

The usage restriction might be employed when a key that could be used for more than one operation is to be restricted.

► Subject alternative name

The subject alternative name extension allows additional identities to be bound to the subject of the certificate. Defined options include:

- An Internet electronic mail address
- A domain name system (DNS) name
- An IP address
- A uniform resource identifier (URI)

The subject alternative name is considered to be definitively bound to the public key.

Example 3-2 shows the main options of a RACF RACDCERT command that you can use to generate a digital certificate.

Example 3-2 RACF command for generating a digital certificate

```

RACDCERT ID(user ID) GENCERT
  SUBJECTSDN(
    CN('common-name')
    T('title')
    OU('organizational-unit-name1',...)
    O('organization-name')
    L('locality')
    SP('state-or-province')
    C('country') )
  SIZE(size-of-new-private-key-in-decimal-bits)
  NOTBEFORE( DATE(yyyy-mm-dd) TIME(hh:mm:ss) )
  NOTAFTER ( DATE(yyyy-mm-dd) TIME(hh:mm:ss) )
  WITHLABEL('label-name')
  SIGNWITH( CERTAUTH|SITE LABEL('label-name') )
  PCICC | ICSF | DSA
  KEYUSAGE( HANDSHAKE DATAENCRYPT DOCSIGN CERTSIGN)
  ALTNAME( IP(numeric-ip-address)
    DOMAIN('internet-domain-name')
    EMAIL('email-address')
    URI('universal-resource-identifier') )

```

The values that you specify for the KEYUSAGE parameter specify the values for the KeyUsage certificate extension as follows:

▶ HANDSHAKE

The key facilitates identification and key exchange during security handshakes, such as SSL. RACF sets the digitalSignature and keyEncipherment indicators in the extension.

▶ DATAENCRYPT

The key is used to encrypt data. RACF sets the dataEncipherment indicator in the extension.

▶ DOCSIGN

The key is used to produce a legally binding signature. RACF sets the nonRepudiation indicator in the extension.

▶ CERTSIGN

The key is used to sign other digital certificates and CRLs. RACF sets the keyCertSign and cRLSign indicators in the extension.

When you choose either PCICC or ICSF, the resulting private key is generated with the RSA algorithm and stored in the ICSF PKDS.

- ▶ If you choose PCICC, the key pair is generated using cryptographic hardware and the resulting private key is stored in the Integrated Cryptographic Services Facility (ICSF) Private Key Data Set (PKDS).
- ▶ If you choose ICSF, the key pair is generated using software and the resulting private key is stored in the ICSF PKDS.

If DSA is specified, the key pair is generated using software with DSA algorithm and the private key is stored in the RACF database as a non-ICSF DSA key. If you omit both, the key pair is generated using software and the private key is stored in the RACF database.

The ICSF PKDS is recommended for the storage of a private key associated with a digital certificate. ICSF ensures that private keys are encrypted under a master key and that access to them is controlled by profiles in the RACF general resource classes CSFKEYS and CSFSERV. See Chapter 4, “Crypto hardware and ICSF” on page 133 for a discussion of cryptographic hardware and ICSF.

3.7.3 Certification paths

In Figure 3-18, certificate authority CA1 issued digital certificate DC1 to certify that public key AKey belongs to Alice. But how do we know that we can trust digital certificate DC1? Well, since DC1 is essentially the message tbsCertificate signed with CA1's private key, we must verify the digital signature just like we verified the digital signature in Figure 3-16 on page 94. That is, if we hold an assured copy of CA1's public key, we must use it to proceed as shown in Figure 3-20.

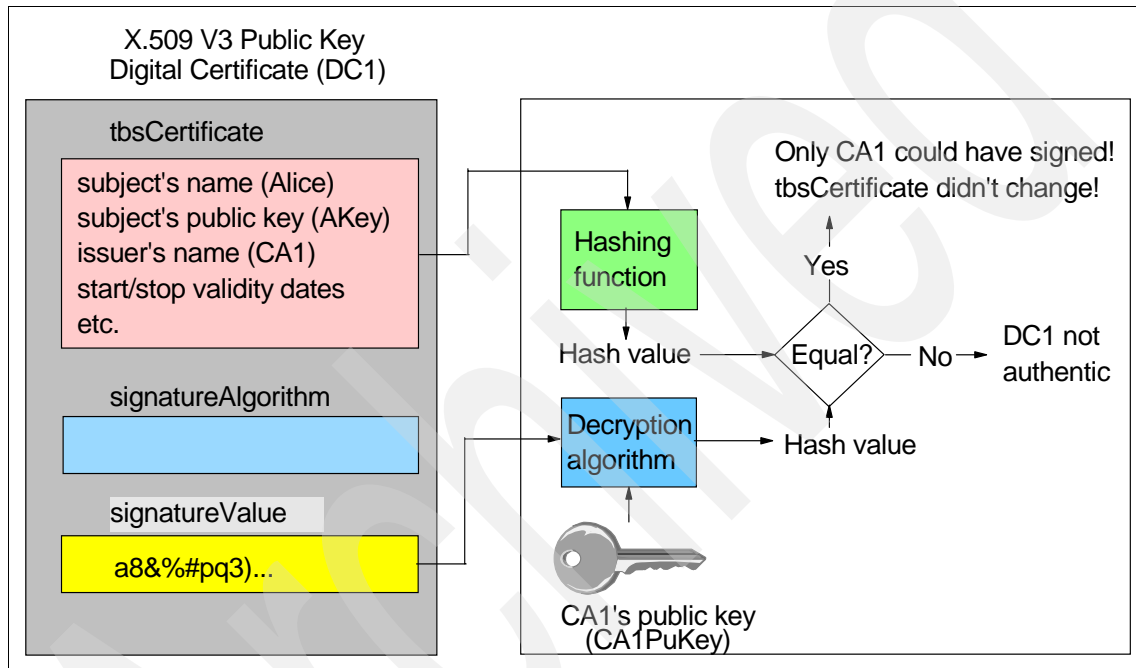


Figure 3-20 Verifying digital certificate

If we do not already hold an assured copy of CA1's public key, then we need a digital certificate signed by another CA to certify that CA1PuKey belongs to CA1. See Figure 3-21.

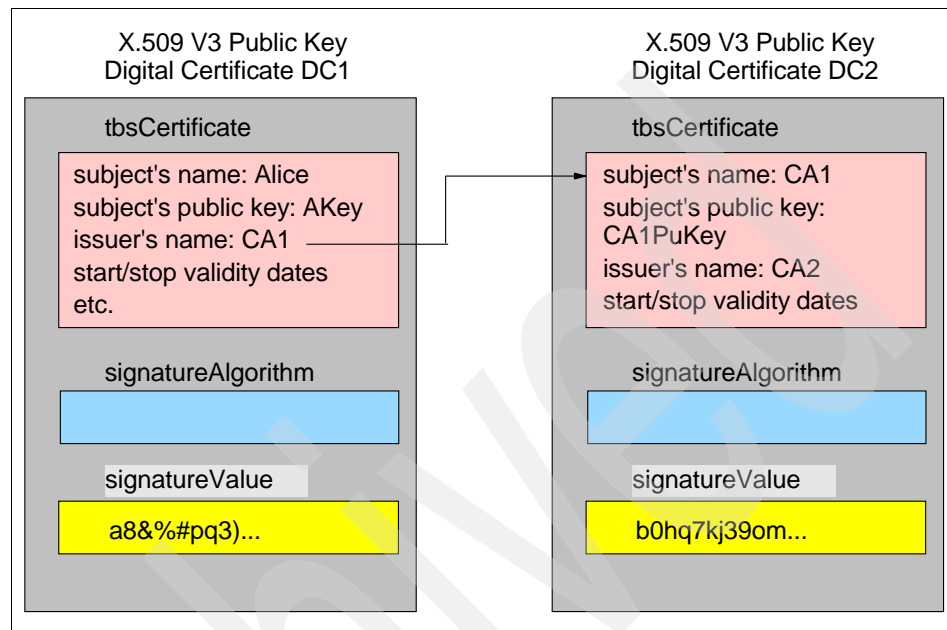


Figure 3-21 Certification path

In general, we need a sequence of n certificates, which satisfies the following conditions:

- ▶ Certificate 1 is the certificate to be validated.
- ▶ For all x in $\{1, 2, \dots, n-1\}$, the issuer of certificate x is the subject of certificate $x+1$.
- ▶ Certificate n is issued by a certificate authority that is trusted without a certificate from any other certifying authority. The certificate can be a self-signed certificate (one in which the CA uses its own private key to attest that the subject public key belongs to the CA).
- ▶ For all x in $\{1, 2, \dots, n\}$ the certificate is valid at the time in question.

Such a sequence is called a *certification path*.

3.8 Certificate revocation lists

When a certificate is issued, it is expected to be in use for its entire validity period. However, various circumstances might cause a certificate to become invalid prior to the expiration of the validity period. Such circumstances include change of name, change of association between subject and CA (for example, an employee terminates employment with an organization), and compromise or suspected compromise of the corresponding private key. Under such circumstances, the CA needs to revoke the certificate.

RFC 3280 defines one method of certificate revocation. This method involves each CA periodically issuing a signed data structure called a *certificate revocation list* (CRL). A CRL is a time-stamped list identifying revoked certificates that is signed by a CA and made freely available in a public repository. Each revoked certificate is identified in a CRL by its certificate serial number. When a certificate-using system uses a certificate, that system not only checks the certificate signature and validity but also acquires a suitably-recent CRL and checks that the certificate serial number is not on that CRL. A new CRL is issued on a regular periodic basis. An entry is added to the CRL as part of the next update following notification of revocation.

Each CRL has a particular scope. The CRL scope is the set of certificates that could appear on a given CRL. For example, the scope could be “all certificates issued by CA X,” “all CA certificates issued by CA X,” or “all certificates issued by CA X that have been revoked for reasons of key compromise and CA compromise.”

A complete CRL lists all unexpired certificates, within its scope, that have been revoked for one of the revocation reasons covered by the CRL scope. The CRL issuer might also generate delta CRLs. A delta CRL only lists those certificates, within its scope, whose revocation status has changed since the issuance of a referenced complete CRL (known as the base CRL).

A CRL is a sequence of three required fields:

- ▶ `tbsCertList`

The `tbsCertList` is itself a sequence of required and optional fields:

- `version` (optional)

The `version` field describes the version of the encoded CRL. When the version is 2, the integer value for the field is 1.

- `signature`

The `signature` field contains the algorithm identifier for the algorithm used to sign the CRL.

- issuer

The issuer field contains an X.500 distinguished name (DN) that identifies the entity that has signed and issued the CRL.

- thisUpdate

This field indicates the issue date of this CRL.

- nextUpdate

This field indicates the date by which the next CRL will be issued. The next CRL could be issued before the indicated date, but it will not be issued any later than the indicated date.

- revokedCertificates (optional)

The revoked certificate list is optional to support the case where a CA has not revoked any unexpired certificates that it has issued. The revokedCertificates field is a sequence of three fields:

- userCertificate

This field contains the serial number of the revoked certificate. Certificates revoked by the CA are uniquely identified by the certificate serial number.

- revocationDate

This field contains the date on which the revocation occurred.

- crlEntryExtensions (optional)

We discuss some of the extensions for CRL entries in 3.8.1, “Extensions for entries in a CRL” on page 111.

- crlExtensions (optional)

We discuss some of the extensions for CRLs in 3.8.2, “Extensions for a CRL” on page 112.

- ▶ signatureAlgorithm

The signatureAlgorithm field contains the algorithm identifier for the algorithm used by the CRL issuer to sign the CRL.

- ▶ signatureValue

The signatureValue field contains a digital signature computed upon the tbsCertList. The ASN.1 DER encoded tbsCertList is used as the input to the signature function. This signature value is encoded as a BIT STRING and included in the CRL signatureValue field.

You can find an example of a certificate revocation list at:

crl.geotrust.com/crls/secureca.crl

3.8.1 Extensions for entries in a CRL

The extensions for CRL entries, which are defined in RFC 3280, include these:

▶ `reasonCode`

This extension identifies the reason for the certificate revocation as follows:

- `unspecified` (0)
- `keyCompromise` (1)
- `caCompromise` (2)
- `affiliationChanged` (3)
- `superseded` (4)
- `cessationOfOperation` (5)
- `certificateHold` (6)
- `removeFromCRL` (8)
- `privilegeWithdrawn` (9)
- `aACompromise` (10)

Apparently the `certificateHold` status is a reversible status that can be used to note the temporary invalidity of the certificate, for instance when the user is not sure if the private key has been lost. If, in this example, the private key was found again and nobody had access to it, the status can be reinstated, and the certificate is valid again, thus removing the certificate from further CRLs.

▶ `holdInstructionCode`

This extension indicates the action to be taken after encountering a certificate that has been placed on hold:

- `reject`
Reject the certificate.
- `callissuer`
Call the certificate issuer or reject the certificate.

▶ `invalidityDate`

This extension provides the date on which it is known or suspected that the private key was compromised or that the certificate otherwise became invalid. This date can be earlier than the revocation date in the CRL entry, which is the date on which the CA processed the revocation. When a revocation is first posted by a CRL issuer in a CRL, the invalidity date can precede the date of issue of earlier CRLs.

3.8.2 Extensions for a CRL

The CRL extensions defined in RFC 3280 include these:

- ▶ `authorityKeyIdentifier`

This extension provides a means of identifying the public key corresponding to the private key used to sign a CRL.
- ▶ `issuerAltName`

This extension allows the following additional identities to be associated with the issuer of the CRL: an electronic mail address, a DNS name, an IP address, and a URI.
- ▶ `cRLNumber`

This extension conveys a monotonically increasing sequence number for a given CRL scope and CRL issuer. It allows users to easily determine when a particular CRL supersedes another CRL. CRL numbers also support the identification of complementary complete CRLs and delta CRLs.
- ▶ `issuingDistributionPoint`

This extension identifies the CRL distribution point and scope for a particular CRL and indicates whether the CRL covers revocation for end entity certificates only, CA certificates only, attribute certificates only, or a limited set of reason codes.
- ▶ `freshestCRL`

This extension identifies how delta CRL information for this complete CRL is obtained.
- ▶ `deltaCRLIndicator`

This extension identifies a CRL as being a delta CRL. Delta CRLs contain updates to revocation information previously distributed, rather than all the information that would appear in a complete CRL. The use of delta CRLs can sometimes reduce network load and processing time. The extension contains the number of the base CRL, that is, it contains the number that identifies the CRL, complete for a given scope, that was used as the starting point in the generation of this delta CRL.

3.8.3 Security considerations when using digital certificates

RFC 3280 advises users to consider the following points when using digital certificates:

- ▶ The procedures performed by CAs to validate the binding of the subject's identity to their public key greatly affect the confidence that ought to be placed in the certificate. Different CAs can issue certificates with varying levels of

identification requirements. One CA might insist on seeing a driver's license, another might want the certificate request form to be notarized, yet another might want fingerprints of anyone requesting a certificate.

Relying parties might want to review the CA's certificate practice statement in order to avoid situations such as the following. Suppose Mallory wants to impersonate Alice. If Mallory can convincingly sign messages as Alice, he can send a message to Alice's bank saying "I want to withdraw \$10,000 from my account. Send me the money." To carry out this attack, Mallory generates a key pair and sends the public key to a CA saying "I'm Alice. Here is my public key. Please send me a certificate." If the CA is fooled and sends him such a certificate, he can then fool the bank.

- ▶ The use of a single key pair for both signature and other purposes is strongly discouraged. Use of separate key pairs for signature and key management provides several benefits to the users. The ramifications associated with loss or disclosure of a signature key are different from loss or disclosure of a key management key. Using separate key pairs permits a balanced and flexible response.
- ▶ The protection afforded private keys is a critical security factor. Failure of users to protect their private keys will permit an attacker to masquerade as them, or decrypt their personal information.
- ▶ The availability and freshness of revocation information affects the degree of assurance that ought to be placed in a certificate. If revocation information is untimely or unavailable, the assurance associated with the binding is clearly reduced.
- ▶ The certification path validation algorithm depends on the certain knowledge of the public keys (and other information) about one or more trusted CAs. The decision to trust a CA is an important decision as it ultimately determines the trust afforded a certificate.
- ▶ The binding between a key and certificate subject cannot be stronger than the cryptographic module implementation and algorithms used to generate the signature. Short key lengths or weak hash algorithms will limit the utility of a certificate.

3.9 Key agreement protocols

A *key agreement protocol*, also called a *key exchange protocol*, is a protocol that allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a secret key algorithm. We discuss two key agreement protocols: RSA and Diffie-Hellman.

3.9.1 The RSA key agreement protocol

In 3.3, “Public key (or asymmetric) cryptography” on page 79 we noted that public key cryptography can be used to safely transmit a secret key. When Alice wants to send a secret key to Bob, she looks up Bob’s public key in a directory, uses it to encrypt the secret key and sends it off to Bob. Bob then uses his private key to decrypt the secret key and read it. No one listening in can decrypt the secret key.

Since RSA is a public key cryptosystem, we can use RSA to safely transmit a secret key. See Figure 3-22.

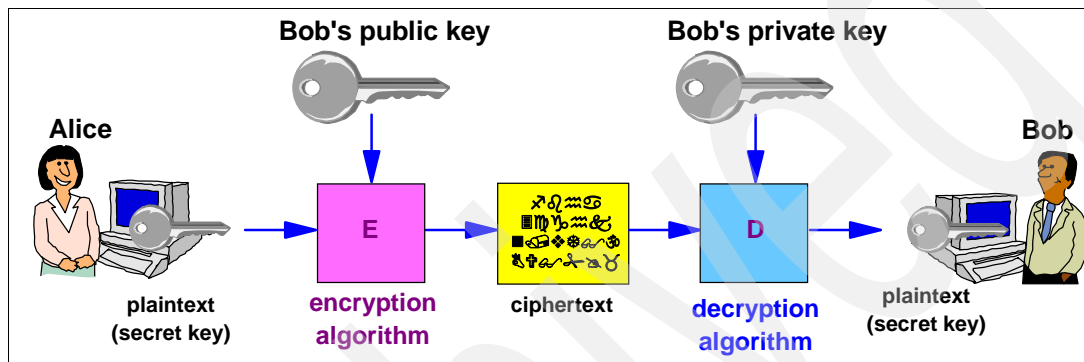


Figure 3-22 RSA key agreement protocol

3.9.2 The Diffie-Hellman key agreement protocol

The Diffie-Hellman key agreement protocol was first published by Whitfield Diffie and Martin Hellman in 1976. The protocol has two system parameters p and g . They are both public and can be used by all the users in a system.

- ▶ Parameter p is a prime number.
- ▶ Parameter g (usually called a generator) is an integer less than p , with the following property: for every integer n between 1 and $p-1$ inclusive, there is a power k of g such that $n = g^k \pmod p$.

For example, if $p = 7$, then $g = 3$ is a generator because: $1 = 3^0 \pmod 7$, $2 = 3^2 \pmod 7$, $3 = 3^1 \pmod 7$, $4 = 3^4 \pmod 7$, $5 = 3^5 \pmod 7$, and $6 = 3^3 \pmod 7$.

Suppose Alice and Bob want to agree on a shared secret key using the Diffie-Hellman key agreement protocol. They proceed as follows:

1. They agree upon a prime number p and a generator g .
2. Alice generates a random private integer a and then derives her public value $g^a \pmod p$.

3. Alice sends her public value to Bob.
4. Bob generates a random private integer b and then derives his public value $g^b \bmod p$.
5. Bob sends his public value to Alice.
6. Alice computes $(g^b)^a \bmod p$.
7. Bob computes $(g^a)^b \bmod p$.

Since $(g^b)^a \bmod p = g^{ba} \bmod p = g^{ab} \bmod p = (g^a)^b \bmod p$, Bob and Alice now have a shared secret key.

As an example, suppose that Alice and Bob agree to use a prime number $p = 23$ and a generator $g = 5$.

1. Suppose Alice chooses a secret integer $a = 6$. She then computes her public value $5^6 \bmod 23 = 8$.
2. Alice sends her public value 8 to Bob.
3. Suppose Bob chooses a secret integer $b = 15$. He then computes his public value $5^{15} \bmod 23 = 19$.
4. Bob sends his public value 19 to Alice.
5. Alice computes $19^6 \bmod 23 = 2$.
6. Bob computes $8^{15} \bmod 23 = 2$.

Alice and Bob now have the shared secret key 2.

The Diffie-Hellman key agreement protocol as just described is vulnerable to a man-in-the-middle attack. In this attack, Eve (for eavesdropper) intercepts Alice's public value and sends her own public value to Bob. When Bob transmits his public value, Eve substitutes it with her own and sends it to Alice. Eve and Alice thus agree on one shared key and Eve and Bob agree on another shared key. After this exchange, Eve simply decrypts any messages sent out by Alice or Bob, and then reads and possibly modifies them before re-encrypting with the appropriate key and transmitting them to the other party. This vulnerability is present because the protocol does not *authenticate* the participants. The protocol as described is sometimes called anonymous Diffie-Hellman.

The authenticated Diffie-Hellman key agreement protocol, or *Station-to-Station* (STS) protocol, was presented by Diffie, van Oorschot, and Wiener in 1992. Prior to execution of this protocol, Alice and Bob each obtain a public/private key pair and a certificate for the public key; they also agree upon the two system parameters p and g . The protocol then proceeds as follows:

1. Alice generates a random number a and computes and sends $g^a \bmod p$ to Bob.

2. Bob generates a random number b and computes $g^b \bmod p$.
3. Bob computes the shared secret key $K = (g^a)^b \bmod p$.
4. Bob concatenates the exponentials $(g^b \bmod p, g^a \bmod p)$ (order is important), signs them using his private key B , and then encrypts them with K . He sends the ciphertext along with his own exponential $g^b \bmod p$ to Alice.
5. Alice computes the shared secret key $K = (g^b)^a \bmod p$.
6. Alice decrypts $(g^b \bmod p, g^a \bmod p)$ using the shared secret key K and verifies Bob's signature using Bob's public key.
7. Alice concatenates the exponentials $(g^a \bmod p, g^b \bmod p)$ (order is important), signs them using her private key A , and then encrypts them with K . She sends the ciphertext to Bob.
8. Bob decrypts and verifies Alice's signature.

Alice and Bob are now mutually authenticated and have a shared secret. This secret, K , can then be used to encrypt further communication.

3.10 Transport Layer Security (TLS) 1.0 protocol

The primary goal of the Transport Layer Security (TLS) protocol is to provide privacy (confidentiality) and data integrity between two applications communicating over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

CICS supports two security protocols that can be used to provide secure communication over the Internet. The first is the Secure Sockets Layer (SSL) 3.0 protocol. The second is the Transport Layer Security (TLS) 1.0 protocol, which is based on SSL 3.0. The TLS 1.0 specification is documented in RFC2246 and is available on the Internet at:

<http://www.rfc-editor.org/rfcsearch.html>

Note: The term SSL is used to refer to both protocols in this book, except where a specific point about either protocol is required.

TLS provides the following enhancements over SSL 3.0:

- ▶ Key-Hashing for Message Authentication

TLS uses Key-Hashing for Message Authentication Code (HMAC), which ensures that a record cannot be altered while travelling over an open network such as the Internet. SSL Version 3.0 also provides keyed message

authentication, but HMAC is considered more secure than the Message Authentication Code (MAC) function that SSL 3.0 uses.

- ▶ Enhanced Pseudorandom Function (PRF)

PRF is used for generating key data. In TLS, the PRF is defined with the HMAC. The PRF uses two hash algorithms in a way that guarantees its security. If either algorithm is exposed, then the data remains secure as long as the second algorithm is not exposed.
- ▶ Improved finished message verification

Both TLS 1.0 and SSL 3.0 provide a finished message to both endpoints that authenticates that the exchanged messages were not altered. However, TLS bases this finished message on the PRF and HMAC values, which is more secure than SSL 3.0.
- ▶ Consistent certificate handling

Unlike SSL 3.0, TLS specifies the type of certificate which must be exchanged between TLS implementations.
- ▶ Specific alert messages

TLS provides more specific and additional alerts to indicate problems that either session endpoint detects. TLS also documents when certain alerts should be sent.

The differences between TLS 1.0 and SSL 3.0 are significant enough that TLS 1.0 and SSL 3.0 do not interoperate (although TLS 1.0 does incorporate a mechanism by which a TLS implementation can back down to SSL 3.0).

3.10.1 TLS overview

The TLS protocol is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. At the lowest level, layered on top of some reliable transport protocol (for example, TCP), is the TLS Record Protocol. The TLS Record Protocol provides connection security that has two basic properties:

- ▶ The connection is private. Secret key cryptography is used for data encryption. The keys for this secret key encryption are generated uniquely for each connection and are based on a secret negotiated by another protocol (such as the TLS Handshake Protocol).
- ▶ The connection is reliable. Message transport includes a message integrity check using a keyed MAC (HMAC). Secure hash functions (for example, SHA-1 or MD5) are used for MAC computations.

The TLS Handshake Protocol operates on top of the TLS Record Protocol and allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol (such as http) transmits or receives its first byte of data. The TLS Handshake Protocol provides connection security that has three basic properties:

- ▶ The peer's identity can be authenticated using public key cryptography.
- ▶ The negotiation of a shared secret is secure: the negotiated secret is unavailable to eavesdroppers, and for any authenticated connection the secret cannot be obtained, even by an attacker who can place himself in the middle of the connection.
- ▶ The negotiation is reliable: no attacker can modify the negotiation communication without being detected by the parties to the communication.

The Record Protocol takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, reassembled, and then delivered to higher level clients. See Figure 3-23, which shows a client sending a message to a server.

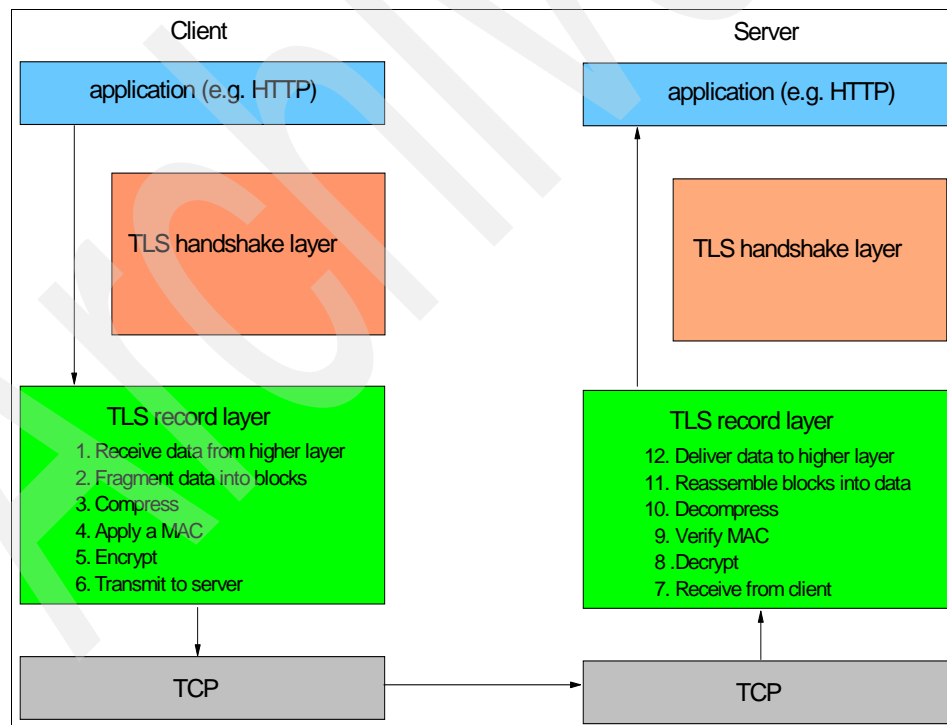


Figure 3-23 Overview of TLS

The TLS Handshake Protocol consists of a suite of three sub-protocols:

- ▶ Change cipher spec protocol
- ▶ Alert protocol
- ▶ Handshake protocol

Before we examine the Handshake protocol, we explain what a *cipher suite* is.

3.10.2 Cipher suites

One dictionary defines a *suite* as “a group of things forming a unit or constituting a collection”. A *cipher suite* then is a collection of cipher algorithms. More specifically, RFC 2246 defines a cipher suite as a collection consisting of one key exchange algorithm, one encryption algorithm, and one hash algorithm.

The hash algorithm must come from the following set:

- ▶ NULL (no hash algorithm)
- ▶ MD5
- ▶ SHA (meaning SHA-1)

The encryption algorithm must come from the following set:

- ▶ NULL (no encryption)
- ▶ IDEA_CBC

IDEA is a 64-bit block cipher designed by Xuejia Lai and James Massey; it uses a 128 bit key. IDEA_CBC is IDEA running in cipher block chaining mode.

- ▶ RC2_CBC_40

RC2 is a variable key-size block cipher designed by Ronald Rivest for RSA Security; it uses a 64-bit block size. RC2_CBC_40 is RC2 running with a 40-bit key in cipher block chaining mode. (“RC” stands for “Ron’s Code” or “Rivest’s Cipher”.)

- ▶ RC4_40

RC4 is a variable key-size stream cipher designed by Rivest for RSA Security. RC4_40 is RC4 running with a 40-bit key.

- ▶ RC4_128

RC4_128 is RC4 running with a 128-bit key. When RFC 2246 was published, RC4_40 was “exportable” but RC4_128 was not. (For many years, the U.S. government did not approve export of cryptographic products unless the key size was strictly limited.)

- ▶ DES40_CBC
DES40_CBC is DES running with a 40-bit key in cipher block chaining mode.
- ▶ DES_CBC
DES_CBC is DES running with a 56-bit key in cipher block chaining mode. When RFC 2246 was published, DES40_CBC was exportable but DES_CBC was not.
- ▶ 3DES_EDE_CBC
3DES_EDE_CBC is TDEA running in cipher block chaining mode. The first use of DES is for encryption, the second for decryption, and the third for encryption.

The key exchange algorithm must come from the following set:

- ▶ DHE_DSS
- ▶ DHE_DSS_EXPORT
- ▶ DHE_RSA
- ▶ DHE_RSA_EXPORT
- ▶ DH_anon
- ▶ DH_anon_EXPORT
- ▶ DH_DSS
- ▶ DH_DSS_EXPORT
- ▶ DH_RSA
- ▶ DH_RSA_EXPORT
- ▶ NULL
- ▶ RSA
- ▶ RSA_EXPORT

DH denotes key exchange algorithms in which the server's certificate contains the Diffie-Hellman parameters signed by the certificate authority. DHE denotes ephemeral Diffie-Hellman, where the Diffie-Hellman parameters are signed by a DSS or RSA certificate, which in turn has been signed by the CA. The signing algorithm used is specified after DH or DHE.

DH_anon indicates completely anonymous Diffie-Hellman communications in which neither party is authenticated. RSA indicates that the server should provide an RSA certificate that can be used for key exchange. (An RSA certificate is an X.509 certificate that has been signed by using the RSA algorithm.)

RFC 2246 assigns names to the cipher suites which contain an acceptable combination of algorithms from the sets identified previously. The names have the form:

TLS_key-exchange-algorithm_WITH_encryption-algorithm_hash-algorithm

For example, the cipher suite TLS_RSA_WITH_DES_CBC_SHA contains the RSA key exchange algorithm, the DES_CBC encryption algorithm, and the SHA hash algorithm. The cipher suite TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA contains the DH_DSS key exchange algorithm, the 3DES_EDE_CBC encryption algorithm, and the SHA hash algorithm.

The TLS 1.0 protocol seeks to provide a framework into which new public key and secret key encryption methods can be incorporated. This prevents the need to create a new protocol (which would risk the introduction of possible new weaknesses). The TLS 1.0 protocol allows additional cipher suites to be registered by publishing an RFC that specifies the cipher suite.

A two digit number is assigned to each cipher suite. In “Enabling CICS to use SSL” on page 197 we provide the list of cipher suites that can be used with CICS TS V3.2 running on a z/OS 1.9 system. Also in “Defining a new TCPIP SERVICE” on page 201 we explain how you can control which cipher suites are used by CICS with the CIPHERS attribute of the TCPIP SERVICE resource definition.

3.10.3 Alert protocol

Error handling in the TLS Handshake protocol is very simple. When an error is detected, the detecting party sends a message called an *alert message* to the other party. An alert message conveys the severity of the message and a description of the alert.

The severity of the message must be one of the following levels:

- ▶ warning (1)
- ▶ fatal (2)

Upon transmission or receipt of a fatal alert message, both parties immediately close the connection. Servers and clients are required to forget any session identifiers, keys, and secrets associated with a failed connection.

The description of the alert must be one of the following possibilities:

- ▶ close_notify (0)
- ▶ unexpected_message (10)
- ▶ bad_record_mac (20)
- ▶ decryption_failed (21)
- ▶ record_overflow (22)
- ▶ decompression_failure (30)
- ▶ handshake_failure (40)
- ▶ bad_certificate (42)
- ▶ unsupported_certificate (43)
- ▶ certificate_revoked (44)

- ▶ `certificate_expired` (45)
- ▶ `certificate_unknown` (46)
- ▶ `illegal_parameter` (47)
- ▶ `unknown_ca` (48)
- ▶ `access_denied` (49)
- ▶ `decode_error` (50)
- ▶ `decrypt_error` (51)
- ▶ `export_restriction` (60)
- ▶ `protocol_version` (70)
- ▶ `insufficient_security` (71)
- ▶ `internal_error` (80)
- ▶ `user_canceled` (90)
- ▶ `no_renegotiation` (100)

RFC 2246 provides a short explanation of each of these descriptions.

The `user-canceled` and `no-renegotiation` alerts carry a level of warning. The sender can determine at its discretion whether the following alerts are fatal or not: `bad_certificate`, `unsupported_certificate`, `certificate_revoked`, `certificate_expired`, `certificate_unknown`, and `decrypt_error`. The remaining alerts always carry a level of fatal.

3.10.4 Handshake protocol

When a TLS client and server first start communicating, they agree on which version of the TLS protocol they will use, select a cipher suite, optionally authenticate each other, and use public key encryption techniques to generate shared secrets.

The TLS Handshake Protocol involves the following steps:

1. Exchange hello messages to agree on a cipher suite and a compression algorithm, exchange random values, and check for session resumption.
2. Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.
3. Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.
4. Generate a master secret from the premaster secret and exchanged random values.
5. Provide security parameters to the record layer as shown in Figure 3-24.
6. Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

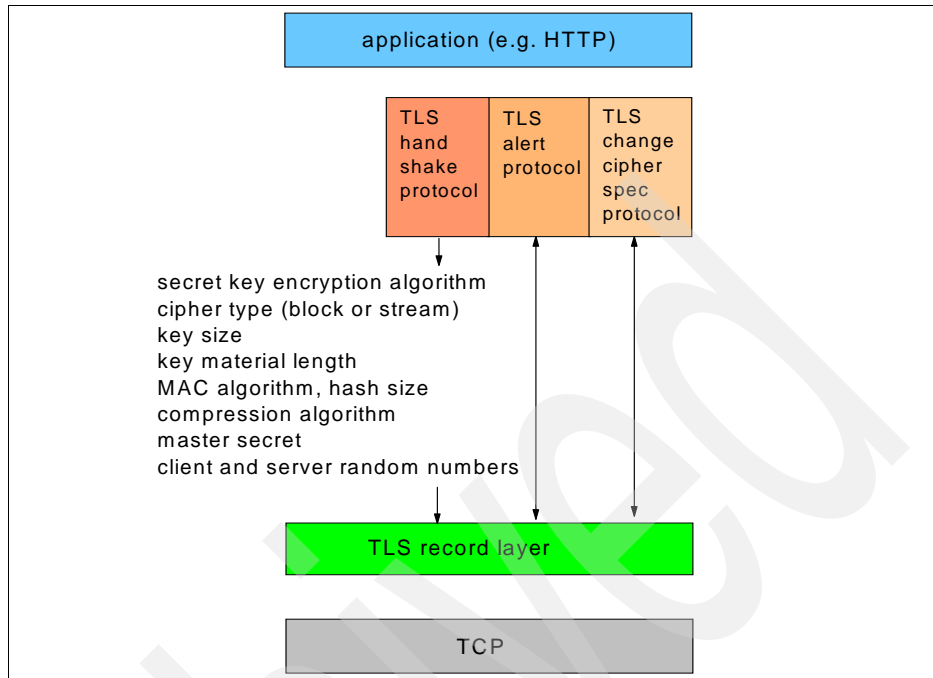


Figure 3-24 Handshake protocol passes security parameters to the record layer

Starting a new session

When the client and server want to start a new session, they begin by exchanging hello messages as shown in Figure 3-25.

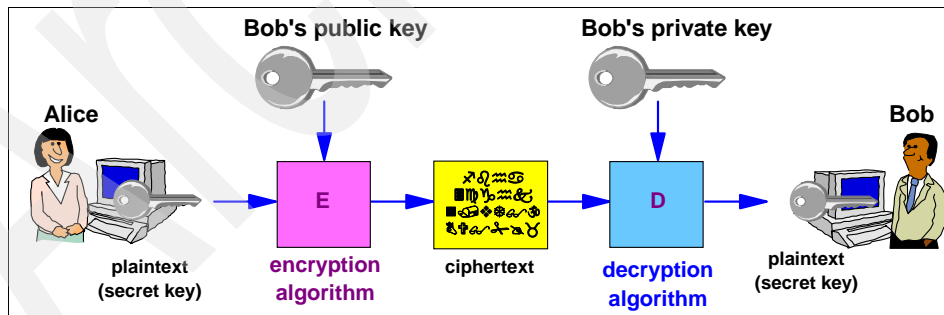


Figure 3-25 Starting a new TLS session(1): Establishing algorithms

The server can send the `hello_request` message at any time. It is a simple notification that the client should begin the negotiation process anew by sending a `client_hello` message when convenient.

The `client_hello` message includes the following parameters:

- ▶ The version of the TLS protocol by which the client wants to communicate during this session. This should be the highest valued version supported by the client. For TLS 1.0 the version should be 3.1. (The version value 3.1 is historical: TLS version 1.0 is a minor modification to the SSL 3.0 protocol, which bears the version value 3.0.)
- ▶ The current time and date according to the client's internal clock, followed by 28 bytes generated by a secure random number generator.
- ▶ A list of the cipher suites supported by the client in order of the client's preference (favorite choice first). Recall that each cipher suite contains a key exchange algorithm, a secret key encryption algorithm (including secret key length), and a MAC algorithm.
- ▶ A list of the compression methods supported by the client, sorted by client preference.

The server will send a `server_hello` message in response to a `client_hello` message when it is able to find an acceptable set of algorithms. If it cannot find such a match, it will respond with a handshake failure alert. The `server_hello` message includes the following parameters:

- ▶ Either the TLS protocol version suggested by the client or the highest TLS protocol version supported by the server, whichever is lower.
- ▶ The current time and date according to the server's internal clock, followed by 28 bytes generated by a secure random number generator.
- ▶ The identity of the session corresponding to this connection. The actual contents of the `sessionID` are defined by the server.
- ▶ The single cipher suite selected by the server from the list supplied by the client.
- ▶ The single compression algorithm selected by the server from the list supplied by the client.

Thus the `client_hello` and `server_hello` messages establish the following connection attributes: the TLS protocol version, the `sessionID`, the key exchange algorithm, the secret key encryption algorithm, the key length for the secret key encryption algorithm, and the compression method. Additionally, two random values are generated and exchanged: `client_hello.random` and `server_hello.random`. Comparing this list of items with the list of items shown in Figure 3-24 on page 123, which the TLS Handshake Protocol must pass to the Record layer, we see that the TLS Handshake Protocol must still come up with a *master secret*.

The master secret is generated by using, among other things, a pre-master secret. The general goal of the key exchange process shown in Figure 3-26 is to create a pre-master secret known to the communicating parties and not to attackers. Note that the italicized lines in Figure 3-26 represent actions taken by the client and server rather than protocol messages.

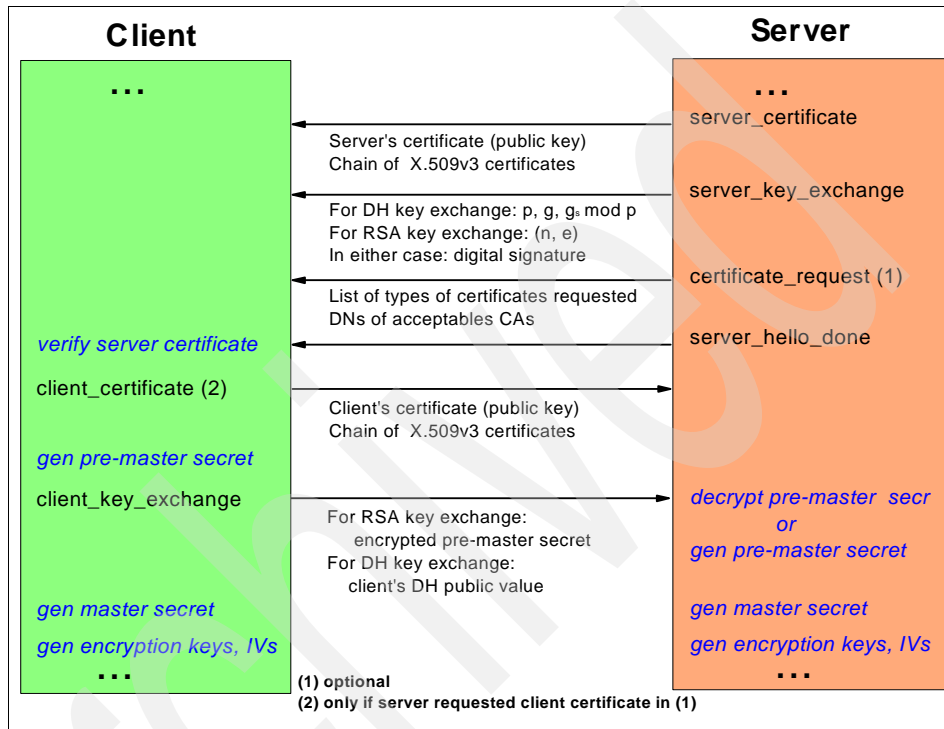


Figure 3-26 Starting a new TLS session(2): Establishing the pre-master secret

The `server_certificate` message sends a chain of X.509v3 certificates. The server's certificate must come first in the chain. Each following certificate must directly certify the one preceding it. The server's certificate must contain a key that matches the key exchange method that was specified in the negotiated cipher suite; see Table 3-2.

Table 3-2 Key exchange methods and corresponding certificate key types

Key exchange method of negotiated cipher suite	Type of key in server certificate
RSA	RSA public key; the keyUsage field of the certificate must allow the key to be used for encryption.
RSA_EXPORT	RSA public key of length greater than 512 bits which can be used for signing, or a key of 512 bits or shorter which can be used for either encryption or signing.
DHE_DSS	DSS public key.
DHE_DSS_EXPORT	DSS public key.
DHE_RSA	RSA public key which can be used for signing.
DHE_RSA_EXPORT	RSA public key which can be used for signing.
DH_DSS	Diffie_Hellman key. The algorithm used to sign the certificate should be DSS.
DH_RSA	Diffie_Hellman key. The algorithm used to sign the certificate should be RSA.

Note: At the time RFC 2246 was written, United States export restrictions limited RSA keys used for encryption to 512 bits, but did not place any limit on lengths of RSA keys used for signing operations.

The `server_key_exchange` message is sent by the server only when the `server_certificate` message does not contain enough data to allow the client to exchange a pre-master secret. This is true for the following key exchange methods: `RSA_EXPORT` (if the public key in the server certificate is longer than 512 bits), `DHE_DSS`, `DHE_DSS_EXPORT`, `DHE_RSA`, `DHE_RSA_EXPORT`, and `DH_anon`.

The `server_key_exchange` message conveys cryptographic information to allow the client to confidentially communicate the pre-master secret to the server: either an RSA public key with which to encrypt the pre-master secret, or a Diffie-Hellman public key with which the client can complete a key exchange (with the result being the pre-master secret).

When the key exchange method is `RSA_EXPORT`, the `server_key_exchange` message includes the following parameters:

- ▶ The modulus n of the server's temporary RSA key.
According to US export law at the time RFC 2246 was written, RSA moduli larger than 512 bits could not be used for key exchange in software exported from the US. This message allows the larger RSA keys encoded in certificates to be used to sign temporary shorter RSA keys.
- ▶ The public exponent e of the server's temporary RSA key.
- ▶ A 36 byte structure of two hashes (one SHA-1 and one MD5), which has been signed with the server's private key. The SHA-1 hash takes as input the concatenation of `client_hello.random`, `server_hello.random`, and (n,e) ; it produces 20 bytes of output. The MD5 hash takes the same input and produces 16 bytes of output.

When the key exchange method is `DHE_DSS`, `DHE_DSS_EXPORT`, `DHE_RSA`, or `DHE_RSA_EXPORT`, the `server_key_exchange` message includes the following parameters:

- ▶ The prime modulus p used for the Diffie-Hellman operation.
- ▶ The generator g used for the Diffie-Hellman operation.
- ▶ The server's Diffie-Hellman public value $g^s \bmod p$.
- ▶ Two integers r and s produced as follows. An SHA-1 hash takes as input the concatenation of `client_hello.random`, `server_hello.random`, p , g , and $g^s \bmod p$; it produces 20 bytes of output. The 20 bytes are run through the Digital Signature Algorithm.

A non-anonymous server can optionally request a certificate from the client. The `certificate_request` message includes the following parameters:

- ▶ A list of the types of certificates requested, sorted in order of the server's preference
- ▶ A list of the distinguished names (DNs) of acceptable certificate authorities (CAs)

The server sends the `server_hello_done` message to indicate that it is done sending messages to support the key exchange, and the client can proceed with its phase of the key exchange. Upon receipt of this message, the client should verify that the server provided a valid certificate and that the certificate has not expired or been revoked. The client should also check that the server hello parameters are acceptable.

The `client_certificate` message is the first message the client can send after receiving the `server_hello_done` message. The client only sends the `client_certificate` message if the server requests a certificate. If the client does not have a suitable certificate to send to the server, it sends a message containing no certificates. If the server requires client authentication in order to continue the handshake, it can respond with a `fatal_handshake` failure alert.

The structure of the `client_key_exchange` message depends on which key exchange method has been selected:

- ▶ If RSA is being used for key agreement and authentication, the client generates a 48-byte pre-master secret, encrypts it using either the public key from the server's certificate or the temporary RSA key provided in a `server_key_exchange` message, and then sends the result in an encrypted pre-master secret message. Since the pre-master secret has been encrypted using the server's public key, the server can decrypt it. In fact, only the server can decrypt it. The pre-master secret consists of two bytes that indicate the latest (newest) version of the TLS protocol supported by the client followed by 46 securely-generated random bytes.
- ▶ If Diffie-Hellman is being used for key agreement, the `client_key_exchange` message conveys the client's Diffie-Hellman public value $g^c \text{ mod } p$. Having the client's Diffie-Hellman public value allows the server to compute the same pre-master secret as the client. (In the event that the key exchange method is `DH_RSA` or `DH_DSS`, and the server requested client certification, and the client was able to respond with a certificate that contained a Diffie-Hellman public key whose group and generator matched those specified by the server in its certificate, then the client will send an empty `client_key_exchange` message.)

Now that the client and the server have agreed upon the pre-master secret they can compute the master secret. For all key exchange methods, the same algorithm is used to convert the pre-master secret into the master secret.

Example 3-3 Computing the master secret

```
master_secret=PRF(pre_master_secret, "master secret",  
                  client_hello.random+server_hello.random)
```

In Example 3-3 PRF is a pseudo-random function defined in RFC 2246, and + represents the concatenation operation. PRF takes as input a secret (such as our pre-master secret), an identifying label (such as "master secret"), and a seed (such as the concatenation of the random numbers generated by the client and the server).

Having computed the master secret, the Record Protocol layer for the client and the Record Protocol layer for the server can now each use PRF to compute a `key_block` as shown in Example 3-4 on page 129.

Example 3-4 Computing the key_block

```
key_block=PRF(master_secret, "key expansion",  
              client_hello.random+server_hello.random)
```

Then the `key_block` is partitioned as follows:

- ▶ `client_write_MAC_secret`
The first `SecurityParameters.hash_size` bytes of the `key_block` become the secret data used to authenticate data written by the client.
- ▶ `server_write_MAC_secret`
The next `SecurityParameters.hash_size` bytes of the `key_block` become the secret data used to authenticate data written by the server.
- ▶ `client_write_key`
The next `SecurityParameters.key_material_length` bytes become the key used to encrypt data written by the client.
- ▶ `server_write_key`
The next `SecurityParameters.key_material_length` bytes become the key used to encrypt data written by the server.
- ▶ `client_write_IV`
The next bytes become the initialization vector for the encryption algorithm when the client encrypts data. The required number of bytes is equal to the block size for block ciphers and zero for stream ciphers.
- ▶ `server_write_IV`
The next bytes become the initialization vector for the encryption algorithm when the server encrypts data.

Note: Since the `client_hello.random` and `server_hello.random` values are unique for each connection, the data encryption keys and MAC secrets will be unique for each connection. Also note that the `server_write_key` and the `client_write_key` are independent of each other.

Figure 3-27 shows the final phase of starting a new TLS session.

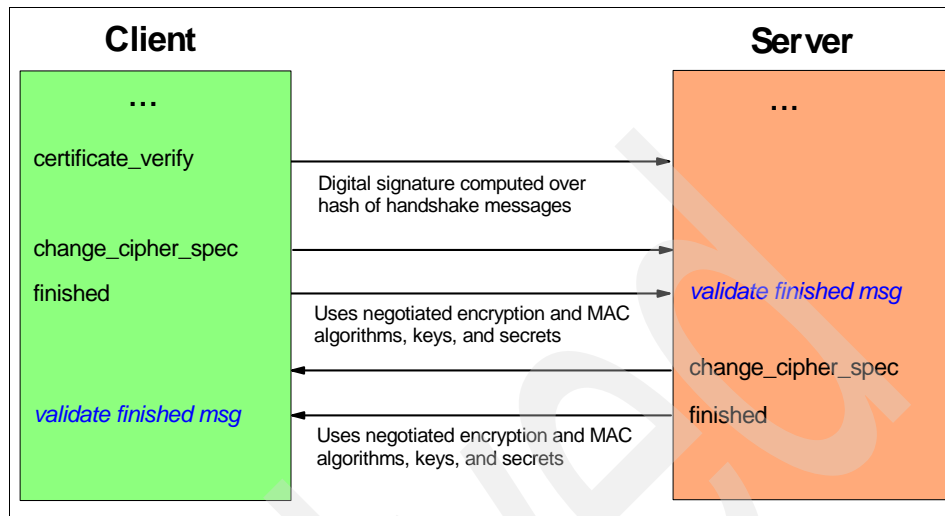


Figure 3-27 Starting a new TLS session (3): Verification

The `certificate_verify` message is used to provide explicit verification of a client certificate. This message is only sent following a client certificate that has signing capability (that is, all certificates except those containing fixed Diffie-Hellman parameters).

When the key exchange method is RSA, the `certificate_verify` message includes a 36-byte structure of two hashes (one SHA-1 and one MD5), which has been signed with the client's private key. The SHA-1 hash takes as input the concatenation of all handshake messages sent or received starting at `client_hello` up to but not including this message; it produces 20 bytes of output. The MD5 hash takes the same input and produces 16 bytes of output. These handshake messages include the server certificate, which binds the signature to the server, and `server_hello.random`, which binds the signature to the current handshake process.

When the key exchange method is Diffie-Hellman, the `certificate_verify` message includes a SHA-1 hash that takes the input described in the preceding paragraph and produces 20 bytes of output. The 20 bytes are then signed using the Digital Signature Algorithm.

The `change_cipher_spec` message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the newly negotiated encryption and MAC algorithms and keys. The message consists of a single byte of value 1.

A finished message is always sent immediately after a `change_cipher_spec` message to verify that the key exchange and authentication processes were successful. The finished message is the first protected with the just-negotiated algorithms, keys, and secrets. The content of the finished message is generated using PRF as shown in Example 3-5.

Example 3-5 Computing the verify_data

```
PRF(master_secret, finished_label,  
    MD5(handshake_messages)+SHA-1(handshake_messages))
```

In Example 3-5:

- ▶ The value of `finished_label` is the string “client finished” for finished messages sent by the client and “server finished” for finished messages sent by the server.
- ▶ The value of `handshake_messages` is all of the data from all handshake messages up to but not including this message.

Recipients of finished messages must verify that the contents are correct. Once a side has sent its finished message and received and validated the finished message from its peer, it can begin to send and receive application data over the connection.

Outgoing data is protected with a MAC before transmission. To prevent message replay or modification attacks, the MAC is computed from the MAC secret, the message contents, the message length, and the sequence number of the message.

Resuming a session

Cryptographic operations tend to be highly CPU intensive, particularly public key operations. For this reason, the TLS protocol has incorporated an optional session caching scheme to reduce the number of connections that need to be established from scratch. When the client and server decide to resume a previous session, the message flow is as shown in Figure 3-28.

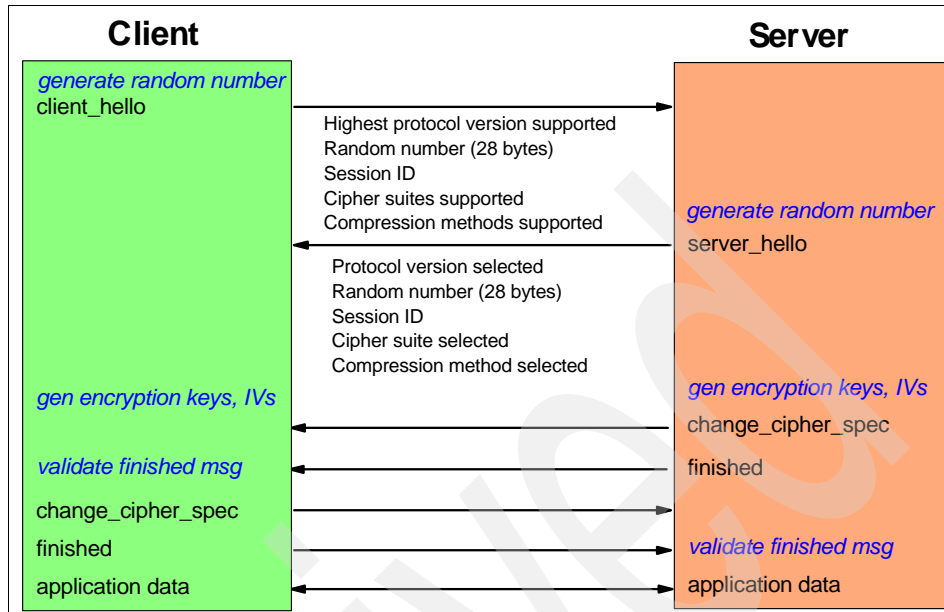


Figure 3-28 Resuming a session

The client sends a `client_hello` using the session ID of the session to be resumed. The server then checks its session cache for a match.

- ▶ If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a `server_hello` with the same session ID value plus the cipher suite and compression method from the state of the session being resumed. At this point both client and server must send `change_cipher_spec` messages and proceed directly to `finished` messages.
- ▶ If a session ID match is not found, the server generates a new session ID and the TLS client and server perform a full handshake.

When a connection is established by resuming a session, new `client_hello.random` and `server_hello.random` values are used with the session's master secret to produce a new `key_block` (see Example 3-4 on page 129) and hence new encryption keys and MAC secrets.

Crypto hardware and ICSF

When using cryptographic functions to secure CICS Web services, it is important to minimize the performance overhead by utilizing hardware cryptographic devices.

In this chapter we introduce you to IBM cryptographic hardware and to the Integrated Cryptographic Service Facility (ICSF), the software product through which the cryptographic hardware is invoked.

We then describe how ICSF and hardware cryptography is used when you implement either the CICS WS-Security support (XML digital signature processing and XML encryption) or CICS support for System SSL.

4.1 Cryptographic hardware

The cryptographic hardware features available to your CICS regions depend on the server that you have. This section provides a summary of the cryptographic hardware features currently available on System z® hardware.

In recent years IBM has shipped the following cryptographic hardware products for mainframe servers:

- ▶ Central Processor Assist for Cryptographic Functions (CPACF)
- ▶ Cryptographic Express 2 Coprocessor (CEX2C)
- ▶ Cryptographic Express 2 Accelerator (CEX2A)
- ▶ Peripheral Component Interconnect Cryptographic Coprocessor (PCICC)
- ▶ Peripheral Component Interconnect Cryptographic Accelerator (PCICA)
- ▶ Peripheral Component Interconnect - Extended Cryptographic Coprocessor (PCIXCC)
- ▶ Cryptographic Coprocessor Feature (CCF)

Table 4-1 shows which of these cryptographic hardware products are available for each of several mainframe servers.

Table 4-1 Cryptographic hardware per server type

Server	9672 G5,G6	z800, z900	z890, z990	z9®, z10™
CPACF	No	No	Yes	Yes
CEX2C (feature 0863)	No	No	Yes	Yes
CEX2A (feature 0863)	No	No	No	Yes
PCIXCC (feature 0868)	No	No	Yes	No
PCICA (feature 0862)	No	Yes (requires CCF)	Yes	No
PCICC (feature 0861)	Yes	Yes (requires CCF)	No	No
CCF (feature 0800)	Yes	Yes	No	No

Next we summarize the main features of the most recent cryptographic hardware options, the CPACF and CEX2.

4.1.1 CP Assist for Cryptographic Functions (CPACF)

CPACF offers a set of symmetric cryptographic functions available on all CPs of a z990, z890, z9-109, z9 Enterprise Class (EC), z9 Business Class (BC), and z10 EC and BC.

The CPACF feature provides hardware acceleration for DES, triple-DES, AES, MAC, and SHA cryptographic services. It provides high-performance hardware encryption, decryption, and hashing support.

The range of services offered by CPACF (the MSA instructions) has increased with each processor family since its introduction on the z990 server. Hence not all services are available on all System z processors. ICSF provides software equivalents for AES encryption and decryption services where hardware facilities are absent on the CPACF. If CPACF functions are invoked via ICSF, then this will be transparent to applications.

Important: The CPACF operates with *clear* keys only. A clear key is a key that has not been encrypted under another key and has no additional protection within the cryptographic environment.

Use of the CPACF instructions provides improved performance. Since the CPACF cryptographic functions are implemented in each PU (processing unit), the potential throughput scales with the number of PUs in the server.

The CPACF feature can be used indirectly by z/OS applications and subsystems (such as CICS) that use the Integrated Cryptographic Service Facility (ICSF) for cryptographic functions (see 4.2, “ICSF” on page 138).

4.1.2 Crypto Express 2 feature

The optional Crypto Express 2 (CEX2) comes as a PCI-X (Peripheral Component Interconnect eXtended) pluggable feature that provides a high performance and secure cryptographic environment. Each CEX2 feature can contain up to two coprocessors, each of which can be configured as an asynchronous cryptographic coprocessor (CEX2C) or accelerator (CEX2A).

- ▶ The Crypto Express 2 *Coprocessor* (CEX2C) feature is designed to secure the cryptographic keys.

Note: A *secure* key is a key that has been encrypted under another key (usually the master key).

Security-relevant cryptographic keys are encrypted under a Master Key when outside of the secure boundary of the CEX2C card. The Master Keys are always kept in battery backed-up memory within the tamper-protected boundary of the CEX2C, and are destroyed if the hardware module detects an attempt to penetrate it. The tamper-responding hardware has been certified at the highest level under the FIPS 140-2 (level 4) standard.

The CEX2C allows the user to do the following tasks, using secure keys:

- Encrypt and decrypt data utilizing shared secret-key algorithms.
- Generate, install, and distribute cryptographic keys securely using both public and secret-key cryptographic methods.
- Generate, verify, and translate personal identification numbers (PINs).
- Ensure the integrity of data by using MACs, hashing algorithms, and RSA public key algorithm (PKA) digital signatures.

Clear key PKA operations are also supported by the CEX2C, and are often used to provide SSL protocol communications.

The CEX2C consolidates the functions previously offered on the z900 by the Cryptographic Coprocessor feature (CCF), the PCI Cryptographic Coprocessor (PCICC), and the PCI Cryptographic Accelerator (PCICA) feature.

- ▶ The Crypto Express 2 *Accelerator* (CEX2A) is actually a CEX2C that has been reconfigured by the user to only provide a subset of the CEX2C functions at enhanced speed.

The CEX2A provides hardware support to accelerate certain cryptographic operations that occur frequently in the e-business environment.

Computationally intensive public key operations as used by the SSL/TLS protocol can be off-loaded from the CP to the CEX2A, potentially increasing system throughput.

The CEX2A is used for the following RSA cryptographic operations (with clear keys only):

- PKA Decrypt (CSNDPKD), with PKCS-1.2 formatting
- PKA Encrypt (CSNDPKE), with ZERO-PAD formatting
- Digital Signature Verify

Important: The CEX2 feature requires ICSF to be active.

A z9 or z10 server can support a maximum of eight CEX2 features. Since each feature can provide two coprocessors or accelerators, a z9 or a z10 server is able to support a maximum of 16 cryptographic coprocessors or accelerators.

4.1.3 Comparison of CPACF, CEX2C, and CEX2A

Table 4-2 summarizes the functions and attributes of the cryptographic hardware that is available for a System z9 and z10.

Table 4-2 Comparison of System z9 and z10 cryptographic hardware

Function or attribute	CPACF	CEX2C	CEX2A
DES/TDES encrypt/decrypt with clear key	X		
AES encrypt/decrypt with clear key	X		
DES/TDES encrypt/decrypt with secure key		X	
Generate random numbers	X	X	
Provide hashing and message authentication	X	X	
Secure key RSA		X	
Clear key RSA		X	X
Provide highest performance for SSL handshaking with clear key			X
Provide highest performance for asymmetric encryption with secure key		X	
Tamper-resistant hardware packaging		X	
Designed for FIPS 140-2 Level 4 certification		X	
Requires ICSF to be active		X	X
Storage for system master keys		X	
Requires system master keys to be loaded		X	
Usable for key management operations		X	

4.1.4 Other cryptographic hardware

In this section we identify the cryptographic hardware that is available with System z servers prior to the z9.

PCI Cryptographic Accelerator (PCICC)

The PCI Cryptographic Coprocessor (PCICC) is an orderable feature that adds additional cryptographic function and cryptographic performance to the z800 and z900 servers, and S/390® G5/G6 servers.

PCI Cryptographic Accelerator (PCICA)

The Peripheral Component Interconnect Cryptographic Accelerator (PCICA) is an orderable feature on the z990 and other zSeries® servers. The PCICA feature is used for the acceleration of modular arithmetic operations, in particular the complex RSA cryptographic operations used with the SSL protocol.

PCI-X Cryptographic Coprocessor (PCIXCC)

The PCIXCC is a single coprocessor card that replaced the CCF and PCICC for the z890 and z990 servers. For more information about the PCIXCC see the article “The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer™” which appeared in volume 48 of the *IBM Journal of Research and Development* for May/July, 2004.

Cryptographic Coprocessor Feature (CCF)

The CCF is a single-chip cryptographic coprocessor that was imbedded as a standard, no-cost component in the early CMOS mainframe systems. Depending on its size, each CMOS mainframe has one or two CCFs. Each CCF contains both DES and Public Key Algorithm (PKA) cryptographic processing units.

For a complete discussion of the System z cryptography infrastructure, refer to *System z Cryptographic Services and z/OS PKI Services*, SG24-7470.

4.2 ICSF

The Integrated Cryptographic Service Facility (ICSF) is a software element of z/OS that works with cryptographic hardware features and RACF to provide secure, high-speed cryptographic services in the z/OS environment. ICSF provides the application programming interfaces by which applications, and subsystems such as CICS, request the cryptographic services.

ICSF provides support for a number of cryptography services, including:

- ▶ DES and triple-DES encryption for privacy
- ▶ The transport of symmetric data keys through the use of the RSA public key algorithm
- ▶ The generation and verification of digital signatures
- ▶ The generation of RSA keys

- ▶ The PKA Encrypt and PKA Decrypt callable services that can be used to enhance the security and performance of SSL/TLS security protocol applications
- ▶ AES encryption and decryption

Note: DSA/DSS operations are no longer supported starting with z990.

4.2.1 ICSF callable services

The format for invoking an ICSF callable service depends on the programming language, for example:

- ▶ C
CSNBxxxx (return_code,reason_code,exit_data_length,exit_data,parameter_5,parameter_6,...,parameter_N)
- ▶ COBOL
CALL 'CSNBxxxx' USING return_code,reason_code,exit_data_length,exit_data,parameter_5,parameter_6,...,parameter_N
- ▶ PL/I
DCL CSNBxxxx ENTRY OPTIONS(ASM);
CALL CSNBxxxx return_code,reason_code,exit_data_length,exit_data,parameter_5,parameter_6,...,parameter_N
- ▶ Assembler
CALL CSNBxxxx,(return_code,reason_code,exit_data_length,exit_data,parameter_5,parameter_6,...,parameter_N)

Note: You cannot use Java to invoke an ICSF callable service, but Java applications can indirectly use the callable services via the IBMJCECCA provider.

Controlling who can use cryptographic keys and services

The ICSF administrator can use RACF to control which applications can use specific keys and services. To set up these controls, the ICSF administrator must create RACF general resource profiles in the CSFKEYS resource class and in the CSFSERV resource class. The CSFKEYS class controls access to cryptographic keys, and the CSFSERV class controls access to most ICSF services.

The following RACF command defines a profile in the CSFKEYS class:

```
RDEFINE CSFKEYS label UACC(NONE) other-optional-operands
```

Where *label* is the label by which the key is defined in the CKDS (Cryptographic Key Data Set) or PKDS (Private Key Data Set).

Use the RACF PERMIT command to give user IDs or groups access to the profile:

```
PERMIT label CLASS(CSFKEYS) ID(groupID) ACCESS(READ)
```

To refresh the in-storage RACF profiles, issue a SETROPTS command:

```
SETROPTS RACLIST(CSFKEYS) REFRESH
```

The following RACF command defines a profile in the CSFSERV class:

```
RDEFINE CSFSERV service-name UACC(NONE) other-optional-operands
```

Where *service-name* is chosen from a list in the *ICSF Administrator's Guide*, SA22-7521.

Use the RACF PERMIT command to give user IDs or groups access to the profile:

```
PERMIT service-name CLASS(CSFSERV) ID(groupID) ACCESS(READ)
```

To refresh the in-storage RACF profiles, issue a SETROPTS command:

```
SETROPTS RACLIST(CSFSERV) REFRESH
```

4.2.2 ICSF administration

You define installation options and configure ICSF using the ICSF panels. You can use the ICSF panels to activate or deactivate your PCICC, PCIXCC, CEX2C, PCICA, and CEX2A coprocessors.

Example 4-1 shows a sample ICSF Coprocessor Management panel. The panel prefixes the coprocessor serial ID with a letter that indicates the type of coprocessor as follows: A for PCICA, E for CEX2C, F for CEX2A, and X for PCIXCC.

Example 4-1 Sample ICSF Coprocessor Management panel

```
----- ICSF Coprocessor Management ----- Row 1 to 4 of 4
```

Select the coprocessors to be processed and press ENTER.

Action characters are: A, D, E, K, R and S. See the help panel for details.

COPROCESSOR	SERIAL NUMBER	STATUS
-----	-----	-----
. E00	95000224	ACTIVE
. E01	95000225	DEACTIVATED
. E02	95000182	DEACTIVATED
. E03	95000180	DEACTIVATED
***** Bottom of data *****		

Note: ICSF recognizes the CEX2A beginning with the FMID HCR7730 level of ICSF. Previous levels of ICSF ignore the CEX2A cards.

4.3 ICSF services used by CICS WS-Security support

The CICS support for XML digital signature processing and XML encryption is dependent on ICSF services and therefore the configuration and startup of ICSF is a requirement for using this support.

Important: ICSF must be started and configured with cryptographic devices in order to use the CICS support for XML digital signature processing and XML encryption.

Figure 4-1 shows how the CICS-supplied message handler, DFHWSSE1, issues a cryptographic API call to the ICSF started task. The ICSF started task invokes RACF to determine whether the user ID associated with the request is authorized to use the requested cryptographic service and any keys associated with the request. If the user ID has the proper authority, the ICSF started task will decide whether it should perform the request using ICSF software or cryptographic hardware.

If ICSF decides to use cryptographic hardware, it gives control to its routines that contain the crypto instructions. If ICSF routes the request to the CEX2C and the request is, for instance, a request to encrypt data, the ICSF started task provides the CEX2C with the data to be encrypted and the key to be used by the encryption algorithm. Recall that the key is encrypted, in this case under a variant of the Symmetric Keys Master Key (SYM-MK) stored in the CEX2C.

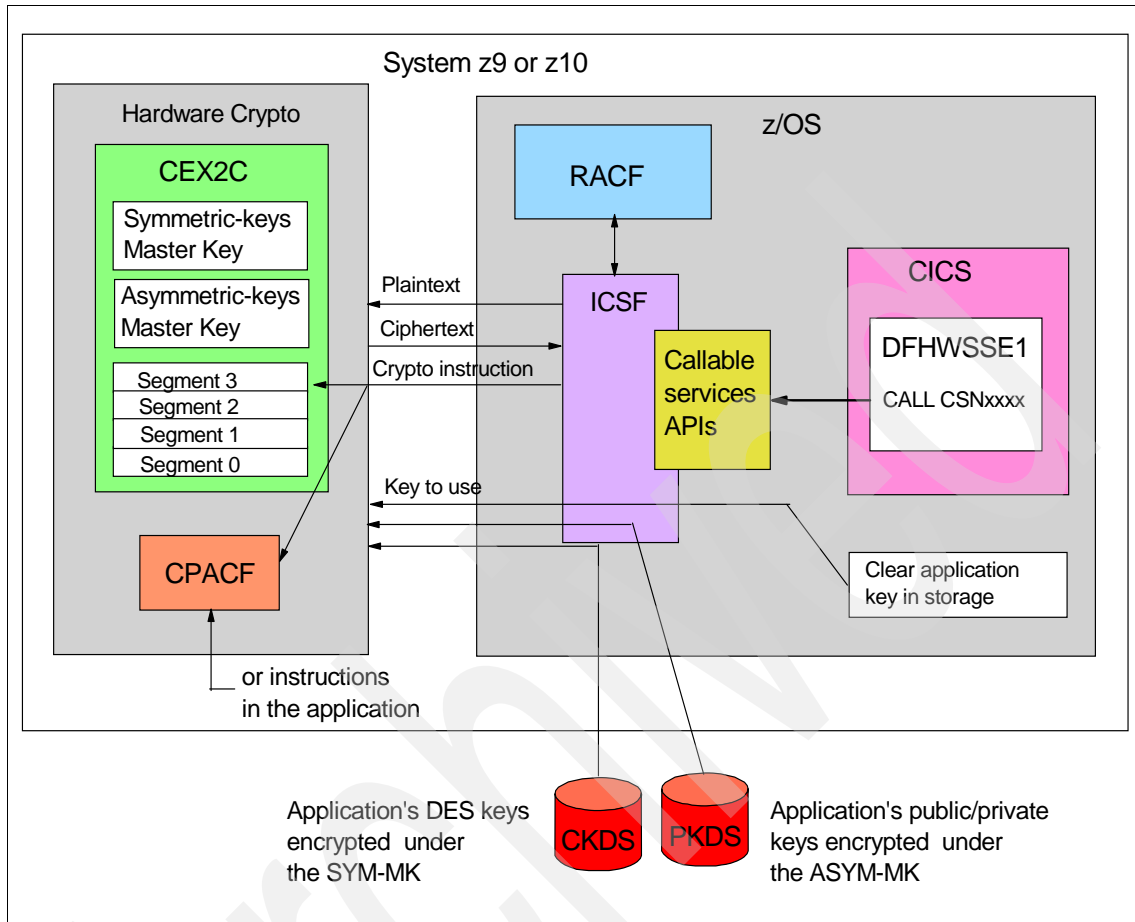


Figure 4-1 Overview of how CICS uses ICSF

The keys can be stored in ICSF-managed VSAM data sets and pointed to by the application program by using the label under which they are stored. The Cryptographic Key Data Set (CKDS) is used to store the symmetric keys in their encrypted form, and the Public Key Data Set (PKDS) is used to store the asymmetric keys.

ICSF callable services

This section describes the ICSF callable services used by the CICS WS-Security support:

- ▶ CSNBCKM
Usage: XML encryption/decryption with DES algorithms

The multiple clear key import callable service imports a clear 64-bit, 128-bit, or 192-bit DATA key that is to be used to encipher or decipher data. This service accepts a clear DATA key, enciphers it under the master key, and returns the encrypted DATA key in operational form in an internal key token.

▶ CSNBDEC

Usage: XML decryption with DES algorithms

The decipher callable service decrypts data in the caller's primary address space using either DES or TDES in the cipher block chaining mode. The caller must supply a 64-byte string that is an internal key token containing the data-encrypting key, or the label of a CKDS record containing a data-encrypting key, to be used for decrypting the data. Thus CSNBDEC uses a secure key. If the key token or CKDS record contains a 64-bit key, a DES decryption is performed. If the key token or CKDS record contains a 128-bit or 192-bit key, TDES decryption is performed.

▶ CSNBENC

Usage: XML encryption with DES algorithms

The encipher callable service encrypts data in the caller's primary address space using either DES or TDES in the cipher block chaining mode. The caller must supply a 64-byte string that is an internal key token containing the data-encrypting key, or the label of a CKDS record containing a data-encrypting key, to be used for encrypting the data. Thus CSNBENC uses a secure key. If the key token or CKDS record contains a 64-bit key, a DES encryption is performed. If the key token or key label contains a 128-bit or 192-bit key, TDES encryption is performed.

▶ CSNBOWH

Usage: XML encryption/decryption and XML digital signature processing

The one-way hash generate callable service generates a one-way hash on specified text. This service supports the following methods: MD5 (software only), SHA-1, RIPEMD-160 (software only), and SHA-256.

▶ CSNBRNG

Usage: XML encryption/decryption and XML digital signature processing

The random number generate callable service generates a 64-bit random number. If the caller requests, the service will generate a number with either even parity or odd parity in each byte. Parity is calculated on the 7 high-order bits in each byte and is presented in the low-order bit in the byte. (Note that a 64-bit random number with odd parity in each byte would be suitable for use as a DES key.)

▶ CSNBSYD

Usage: XML decryption

The symmetric key decipher callable service decrypts data in the caller's primary address space using one of the following algorithms: DES,

TDES-128, TDES-192, AES-128, AES-192, AES-256. The caller can specify the use of either the electronic code book mode or the cipher block chaining mode. The caller must supply a key to be used for decrypting the data. The key must be supplied in one of the following ways: a clear key, a 64-byte string that is an internal key token containing a clear key, or a 64-byte string that is the label of a CKDS record containing a clear key.

▶ CSNBSYE

Usage: XML encryption

The symmetric key encipher callable service encrypts data in the caller's primary address space using one of the following algorithms: DES, TDES-128, TDES-192, AES-128, AES-192, AES-256. The caller can specify the use of either the electronic code book mode or the cipher block chaining mode. The caller must supply a key to be used for encrypting the data. The key must be supplied in one of the following ways: a clear key, a 64-byte string that is an internal key token containing a clear key, or a 64-byte string that is the label of a CKDS record containing a clear key.

▶ CSNDDSG

Usage: XML digital signature generation

The digital signature generate callable service generates a digital signature using an RSA private key. The private key must be valid for signature usage. The caller must supply input text that has been previously hashed using the one-way hash generate callable service.

▶ CSNDDSV

Usage: XML digital signature validation

The digital signature verify callable service verifies a digital signature using an RSA public key. The caller must supply input text that has been previously hashed using the one-way hash generate callable service. The caller must also supply the digital signature that is to be verified.

▶ CSNDPKB

Usage: XML digital signature validation

The PKA key token build callable service can be used to do one of the following:

- Take a clear public key and a clear private key as input and build a PKA private external key token that contains a clear public key and a clear private key. You can use this token as input to the PKA key import service to obtain a private internal key token containing an enciphered private key.
- Take a clear public key as input and build a PKA public external key token containing a clear public key. You can use this token directly in other PKA services.

A “PKA key” means either an RSA key or a DSS key.

- ▶ CSNDPKD
Usage: XML decryption
The PKA decrypt service decrypts a formatted key, deformats it, and returns the deformatted value to the application in the clear.
- ▶ CSNDPKE
Usage: XML encryption
The PKA encrypt service encrypts a supplied clear key value under an RSA public key.

Cryptographic hardware required

The *z/OS ICSF Application Programmer's Guide*, SA22-7522 provides details about the cryptographic hardware required by each callable service for a given server model.

Table 4-3 shows the cryptographic hardware used by ICSF for each callable service listed in the previous section.

Table 4-3 Cryptographic hardware used by ICSF callable services

Service	IBM eServer zSeries 800 or 900	IBM eServer zSeries 890 or 990	IBM System z9-109	IBM System z9 or z10
CSNBCKM	CCF	PCIXCC CEX2C	CEX2C	CEX2C
CSNBDEC	CCF	PCIXCC CEX2C	CEX2C	CEX2C
CSNBENC	CCF	PCIXCC CEX2C	CEX2C	CEX2C
CSNBOWH	SHA-1 requires CCF (SHA-256 not supported by CCF)	SHA-1 requires CPACF (SHA-256 not supported by CPACF)	SHA-1 and SHA-256 require CPACF	SHA-1 and SHA-256 require CPACF (z10 also supports SHA-384 and SHA-512)
CSNBRNG	CCF	PCIXCC CEX2C	CEX2C	CEX2C
CSNBSYD	(Software decryption only)	CPACF	CPACF	CPACF
CSNBSYE	(Software encryption only)	CPACF	CPACF	CPACF
CSNDDSG	CCF PCICC	PCIXCC CEX2C	CEX2C	CEX2C

Service	IBM eServer zSeries 800 or 900	IBM eServer zSeries 890 or 990	IBM System z9-109	IBM System z9 or z10
CSNDDSV	CCF	PCICA PCIXCC CEX2C	CEX2C CEX2A	CEX2C CEX2A
CSNDPKB	None	None	None	None
CSNDPKD	CCF PCICC PCICA	PCICA PCIXCC CEX2C	CEX2C CEX2A	CEX2C CEX2A
CSNDPKE	CCF PCICC	PCICA PCIXCC CEX2C	CEX2C CEX2A	CEX2C CEX2A

To conclude our review of how hardware cryptography is used with the CICS WS-Security support, we list below the z9 and z10 system requirements:

- ▶ Feature 3863 must be installed.
- ▶ At least one Feature 0863 (CEX2) must be installed.
- ▶ At least one of the cards of the CEX2 feature must be configured as a coprocessor (CEX2C) rather than an accelerator (CEX2A).
- ▶ The operating system level must be at z/OS 1.6 or higher.
- ▶ The ICSF level must be at FMID HCR7730 or higher.
- ▶ ICSF must be active.
- ▶ The CEX2C must be *online* to z/OS.
- ▶ The CEX2C must be *active* to ICSF.
- ▶ The user ID under which the CICS region runs needs READ access to the RACF profiles in the CSFSERV class which protect the callable services shown in “ICSF callable services” on page 142.

Furthermore, we recommend that you specify API(OPENAPI) on the PROGRAM definition for the DFHWSSE1 program so that CICS executes the program on an open TCB.

4.4 ICSF services used by System SSL

CICS uses System SSL to support both the SSL 3.0 and TLS 1.0 protocols. System SSL, in turn, makes use of ICSF services and hardware cryptographic if available.

This section provides information about which ICSF services are used by System SSL and the cryptographic hardware that System SSL supports.

System SSL handshake processing utilizes both RSA encryption and digital signature functions. These functions are very expensive functions when performed in software. For installations that have high volumes of SSL handshake processing, utilizing the capabilities of the hardware will provide maximum performance and throughput, and it will also reduce CPU costs.

For installations that are more concerned with the transfer of encrypted data than with SSL handshakes, moving the encrypt/decrypt processing to hardware will provide maximum performance. The encryption algorithm is determined by the SSL cipher suite.

To utilize hardware, the cipher suite's encryption algorithm must be available in hardware. For example, on a z9-109, if you specify a cipher suite that uses TDES to encrypt/decrypt data, then you will benefit from the processing being done in the hardware (using the CPACF). On the other hand, if you specify a cipher suite that uses AES-256 to encrypt/decrypt data, then the processing will be done in software.

In Table 4-4 each row represents a cryptographic algorithm or function that System SSL supports, and each column represents a cryptographic device. If an X appears at the intersection of a row and a column, then System SSL is able to implement the algorithm or function represented by the row using the cryptographic device.

Note: The ability to use a specific cryptographic device for a particular cryptographic algorithm depends on the specific server that is being used.

Table 4-4 Hardware cryptographic functions used by System SSL

Algorithm or function	CCF	PCIXCC	CPACF	CEX2C	CEX2A
RC2					
RC4					
DES	X		X		
TDES	X		X		
AES-128			X		
AES-192			X		
AES-256			X		

MD5					
SHA-1			X		
SHA-256			X		
SHA-512			X		
PKA (RSA) Decrypt	X	X		X	X
PKA (RSA) Encrypt	X	X		X	X
Digital Signature Generate	X	X		X	
Digital Signature Verify	X	X		X	X

In order for System SSL to use the hardware support provided through ICSF, the ICSF started task must be running prior to CICS initialization and the CICS user ID must be authorized to the appropriate resources in the CSFSERV class, if defined (see “Controlling who can use cryptographic keys and services” on page 139).

Table 4-5 identifies the required CSFSERV resource class accesses for different cryptographic algorithms and functions.

Table 4-5 CSFSERV resource class access

Function	z800, z900	z890, z990, z9-109, z9 BC, z9 EC, z10 EC
DES	CSFCKI, CSFDEC, CSFENC	
TDES	CSFCKM, CSFDEC, CSFENC	
PKA (RSA) Decrypt	CSFPKD	CSFPKD
PKA (RSA) Encrypt	CSFPKE	CSFPKE
Digital Signature Generate	CSFPKI, CSFDSG	CSFPKI, CSFDSG
Digital Signature Verify	CSFDSV	CSFDSV

The resource classes are as follows:

- ▶ CSFCKI - Clear key import
- ▶ CSFCKM - Multiple clear key import
- ▶ CSFDEC - Symmetric key decrypt
- ▶ CSFDSG - Digital signature generate
- ▶ CSFDSV - Digital signature verify
- ▶ CSFENC - Symmetric key encrypt
- ▶ CSFPKD - PKA decrypt
- ▶ CSFPKE - PKA encrypt
- ▶ CSFPKI - PKA key import

In addition to the CSFSERV class, the CICS user ID requires access to the RACF CSFKEYS class when key rings are being used and the certificate keys are stored in an ICSF data set.

Archived



Security scenarios environment

In this chapter we provide an overview of the scenarios documented in the subsequent chapters.

We also describe the basic configuration tasks required to run the scenarios, including these:

- ▶ Customizing CICS system initialization parameters
- ▶ Creating basic RACF definitions
- ▶ Creating CA certificates
- ▶ Modifying wsbind files
- ▶ Installing the Catalog example application
- ▶ Testing the environment

5.1 Scenarios: A high level overview

In this section we provide a high level overview of the scenarios that are configured and tested in the following chapters. These scenarios are:

- ▶ Enabling SSL:
SSL client authentication between service requester and service provider.
- ▶ Signing the SOAP message:
Using an XML digital signature to sign a SOAP message.
- ▶ Identity assertion with WebSphere for z/OS:
Asserting an identity from service requester to service provider.
- ▶ Identity assertion with WebSphere DataPower:
Asserting an identity from intermediary server to service provider.
- ▶ Enabling WS-Trust with TFIM:
Using a Security Token Service to transform one identity token into another.

For each of the scenarios that we document, a WebSphere Application Server application acts as a service requester (or consumer) and the CICS Catalog example application is the service provider. The Catalog application runs in a CICS TS V3.2 server. The test scenarios are based on common customer scenarios. Other security scenarios (based on CICS TS V3.1) are documented in the Redbooks publication, *Implementing CICS Web Services*, SG24-7206.

Enabling SSL

In this section we use SSL to encrypt the a SOAP message and SSL client authentication to establish trust between the Web service requester and provider.

This scenario is shown in Figure 5-1.

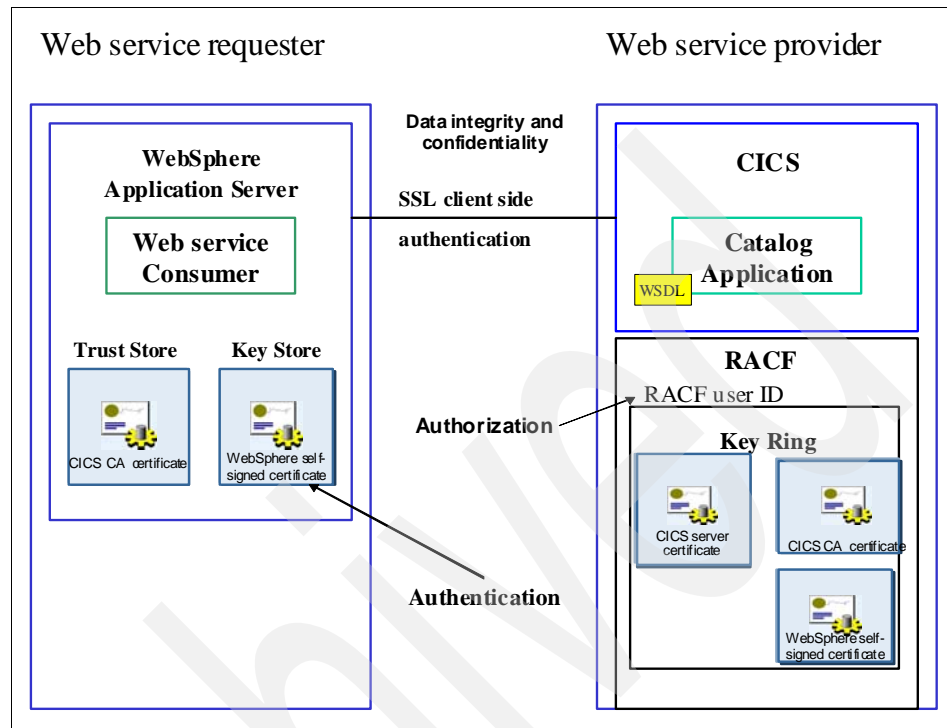


Figure 5-1 Enabling SSL

The sequence of steps for this scenario is as follows:

1. The Web service requester sends an encrypted service request using HTTPS.
2. CICS initiates an SSL client authenticated handshake and an SSL connection is established.
3. CICS creates a security context with a mapping of the Distinguished Name in the certificate (that WebSphere sends as an SSL client certificate) to a RACF user ID.
4. The service provider transaction runs under the mapped RACF user ID and sends an encrypted response to the service requester.

We test this scenario in Chapter 6, “Enabling SSL” on page 181.

Signing the SOAP message

In this scenario, the Web service requester uses an XML digital signature to sign the SOAP message.

This scenario is shown in Figure 5-2.

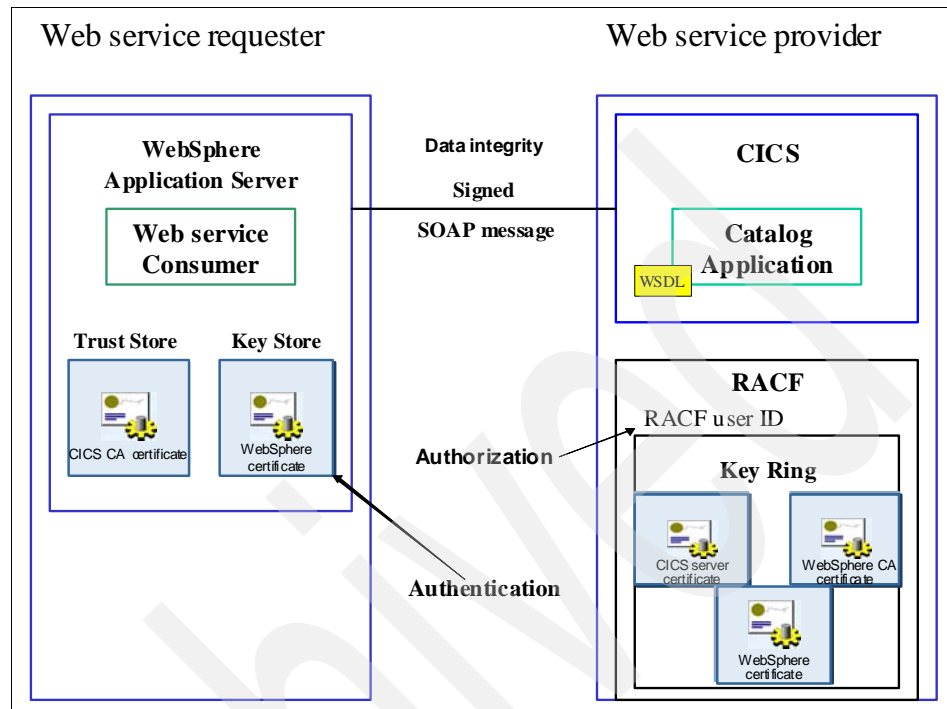


Figure 5-2 Signing the SOAP message.

The sequence of steps for this scenario is as follows:

1. The Web service requester signs the SOAP message, using an XML digital signature and sends the request using HTTP.
2. CICS validates the signature in the request message.
3. CICS creates a security context with a mapping of the Distinguished Name in the certificate (that is used to sign the message) to a RACF user ID.
4. The service provider transaction runs under the mapped RACF user ID
5. CICS signs the response message before sending it to the service requester.
6. WebSphere validates the signature in the response message.

Note: It is assumed that there is no requirement for encryption in this scenario.

We test this scenario in Chapter 7, “Signing the SOAP message” on page 221.

Identity assertion with WebSphere for z/OS

In this scenario, WebSphere Application Server running on z/OS validates the end user's identity. WebSphere asserts the identity (as a RACF user ID) when the service requester application sends a service request to the CICS service provider application.

This scenario is shown in Figure 5-3.

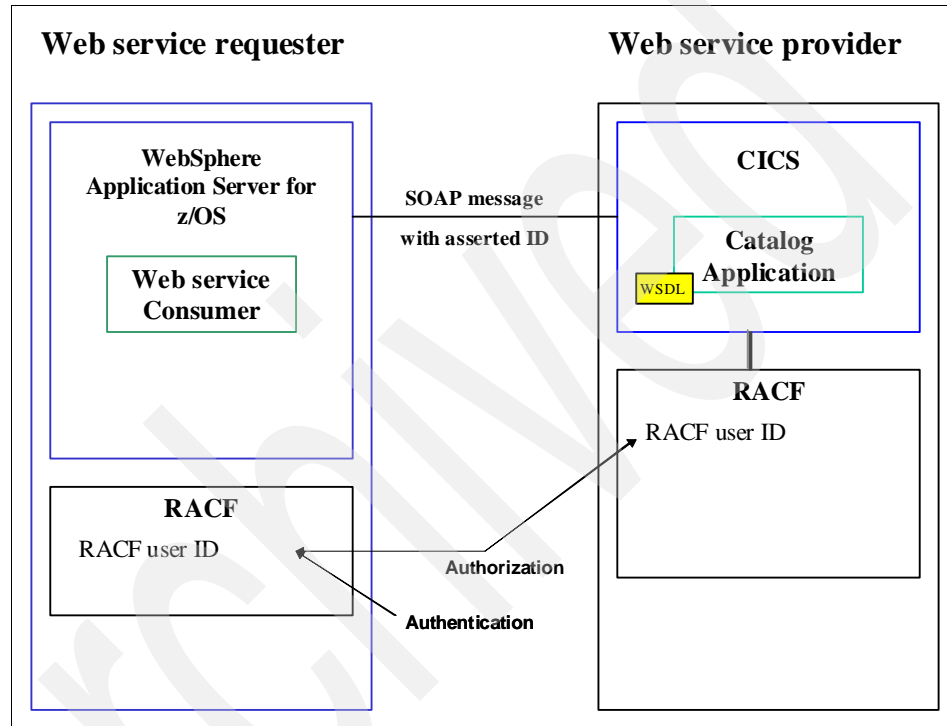


Figure 5-3 Identity assertion with WebSphere for z/OS.

The sequence of steps for this scenario is as follows:

1. The end user's identity is authenticated by WebSphere Application Server for z/OS, such that the WebSphere application runs under the user's RACF ID.
2. When the service request is sent from Web service requester application, WebSphere Application Server attaches a SOAP header containing the RACF user ID to the request message.
3. CICS receives the request and runs the service provider transaction under the RACF user ID received in the SOAP header.

Note: It is assumed that there is no requirement for data integrity or encryption of the SOAP message in this scenario. If this is not the case, an SSL connection could be used between WebSphere Application Server and CICS.

We test this scenario in Chapter 8, “Identity assertion with WebSphere for z/OS” on page 259.

Identity assertion with WebSphere DataPower

In this scenario, the Web service requester uses an XML digital signature to sign the SOAP message. WebSphere DataPower is used to validate the XML digital signature. WebSphere DataPower then asserts the X509 certificate (that is contained in the original request message) to CICS, in an unsigned message.

This scenario is shown in Figure 5-4.

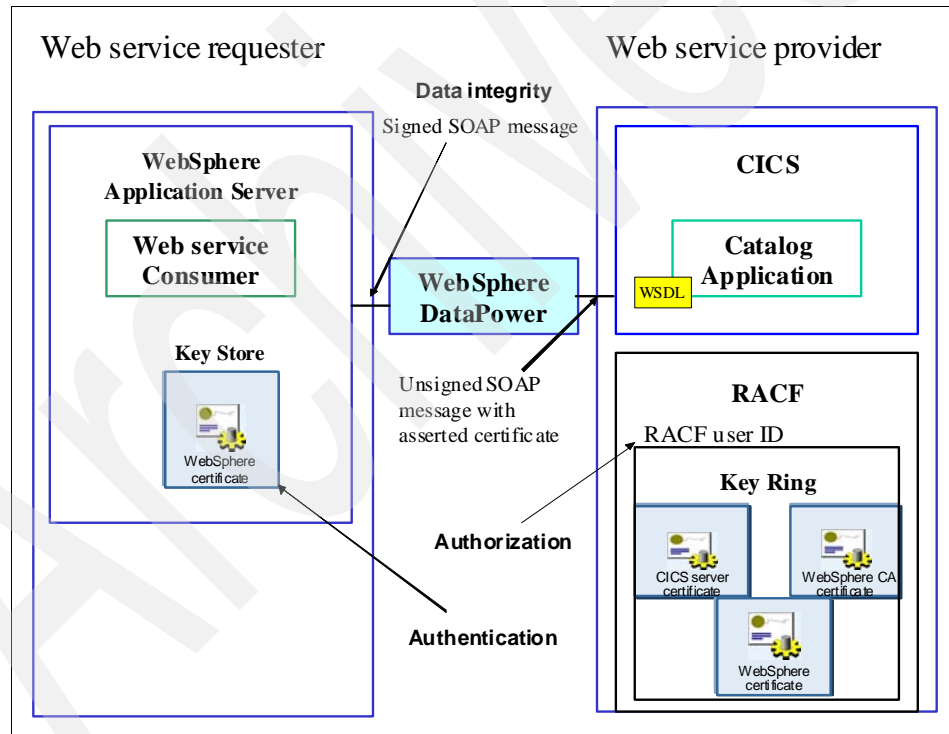


Figure 5-4 Identity assertion with WebSphere DataPower

The sequence of steps for this scenario is as follows:

1. The Web service requester signs the SOAP message, using an XML digital signature, and sends the request using HTTP.
2. WebSphere DataPower validates the XML digital signature, strips the signature from the message, and forwards the X.509 certificate (that is contained in the original request message) to CICS in an unsigned message.
3. CICS creates a security context with a mapping of the Distinguished Name in the certificate to a RACF user ID.
4. The service provider transaction runs under the mapped RACF user ID.
5. CICS sends the (unsigned) response message to WebSphere DataPower, which in turns forwards it to the service requester.

Note: It is assumed that there is no requirement for encryption in this scenario.

We test this scenario in Chapter 9, “Identity assertion with WebSphere DataPower” on page 281.

Enabling WS-Trust with TFIM

In this scenario, we use Tivoli Federated Identity Manager (TFIM) to map a user identity token received from the Web services requester into a RACF user ID. The Web service provider transaction then runs under this user ID. We test with two token types, a UsernameToken and an LTPA token.

This scenario is shown in Figure 5-5.

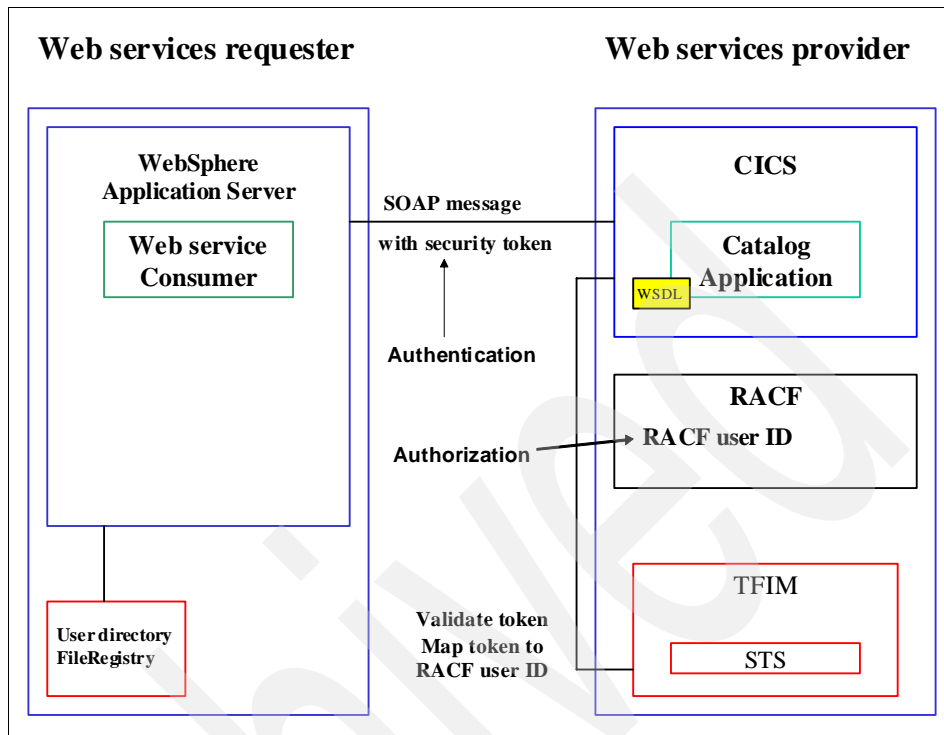


Figure 5-5 Enabling WS-Trust with TFIM

The sequence of steps for this scenario is as follows:

1. When the service request is sent from Web service requester application, WebSphere Application Server attaches a SOAP header containing a security token (either a UsernameToken or an LTPA token).
2. CICS receives the request and calls the Security Token Service (in this case TFIM) to validate the token and map the token to a RACF user ID.
3. CICS runs the service provider transaction under the RACF user ID received from TFIM.
4. CICS sends the response message to the service requester.

Note: It is assumed that there is no requirement for data integrity or encryption of the SOAP message in this scenario. If this is not the case, an SSL connection could be used between WebSphere Application Server and CICS, and between CICS and TFIM.

We test this scenario in Chapter 10, “Enabling WS-Trust with TFIM” on page 313.

5.2 Preparation

In this section we describe the basic CICS and RACF security configuration tasks required before starting to configure the specific security scenarios.

5.2.1 Software checklist

The software that we use is listed in Table 5-1.

Table 5-1 Software used in the security scenarios.

z/OS	Windows®
zOS V1.9	Windows XP SP3
CICS Transaction Server V3.2	WebSphere Application Server V 6.1.0.17

5.2.2 Definition checklist

The main CICS definitions that we use are listed in Table 5-2.

Table 5-2 Settings used in the security scenarios

Value	CICS TS
IP name	wtsc66.itso.ibm.com
TCP/IP port	1430n
SSL/TLS port	1431n
Jobname	CIWSS3Cn
APPLID	A6POS3Cn
TCPIPSERVICE	S3C1
Provider PIPELINE	EXPIPE01
Configuration files	ITSO_7658_basicssoap12provider.xml

In Table 5-2, n is a variable 1-5, which depends on the specific CICS region that is used for the test scenario.

The CICS Web services that we use in the scenarios are listed in Table 5-3. These services are provided as part of the CICS Catalog manager example application (see “The CICS catalog manager example application” on page 174).

Table 5-3 Web services used in the scenarios

Web service	CICS transaction ID	User ID
inquireSingle	INQS	PUBLIC01
inquireCatalog	INQC	PUBLIC01
placeOrder	ORDR ORDS	PRIVAT01 Web service requester identity

We have the following authorization requirements:

- ▶ We want the inquireSingle service to run with transaction ID INQS and under a fixed user ID PUBLIC01.
- ▶ We want the inquireCatalog service to run with transaction ID INQC and under a fixed user ID PUBLIC01.
- ▶ We want the placeOrder service to run initially with transaction ID ORDR and under a fixed user ID PRIVAT01. We then intend to switch the security context for this service so that the target business logic program runs under the Web service requester's identity. In order to switch the user ID, CICS needs to start a second task, and we want this task to run with transaction ID ORDS (a secure Order).

The full set of CICS user IDs used in our scenario are listed in Table 5-4.

Table 5-4 CICS User IDs

Value	CICS TS
CICS region user ID	CIWS3D
CICS default user ID	CICSUSER
User IDs for inquire services (inquireSingle and inquireCatalog)	PUBLIC01
User IDs for order service (placeOrder)	PRIVAT01 USERWS01 USERWS02 USERWS03 USERWS05

The full set of X.509 certificates that we use in our scenarios are listed in Table 5-5.

Table 5-5 Certificates

Value	Format	File
CICS server certificate	PKCS12DER	CIWSS3C1.P12 CIWSS3C2.P12
CICS CA certificate	CERTDER	CIWSROOT.CER
WebSphere CA certificate	CERTDER	WASWINROOT.CER
WebSphere server certificate (CA-signed)	PKCS12DER	WWSERV01.P12
WebSphere server certificate (self-signed)	CER	waswincert.cer
WebSphere z/OS CA certificate	CERTDER	WZOSROOT.CER
WebSphere z/OS server certificate	PKCS12DER	WASZOSS1.P12

5.3 Basic security configuration

First we discuss our basic security configuration, taking a CICS region with no security and configuring it to enable transaction security (we do not implement other types of CICS security such as resource security and command security). Then we document the system initialization table parameters necessary to set up basic CICS security and then we test our basic security configuration.

For detailed information about CICS security, see *CICS Transaction Server for z/OS RACF Security Guide*, SC34-6454.

5.3.1 Setting up basic security configuration

We configured our CICS region with security prefixing, transaction security, and surrogate user security active using the following SIT parameters:

- ▶ SEC=YES
- ▶ SECPRFX=YES
- ▶ XTRAN=YES
- ▶ XUSER=YES

5.3.2 SIT parameters

We specified SEC=YES to indicate that we wanted RACF services to control access to CICS resources.

We used security prefixing (SECPRFX=YES) in our CICS region, which prevents our RACF security profiles from affecting other CICS regions. This is useful in a production environment because it means that all security profiles are unique to an individual region; however, it can mean more work for the security administrator because more profiles must be defined.

XTRAN=YES was specified so CICS would control who could start transactions and XUSER=YES specifies that CICS is to perform surrogate user checking.

5.4 Creating CA certificates

In this section we describe how we create the CA certificates used in the test scenarios.

Note: The creation of the client and server certificates used in the test scenarios is described in the specific scenario chapters.

The total set of certificates and their relationship is shown in Figure 5-6.

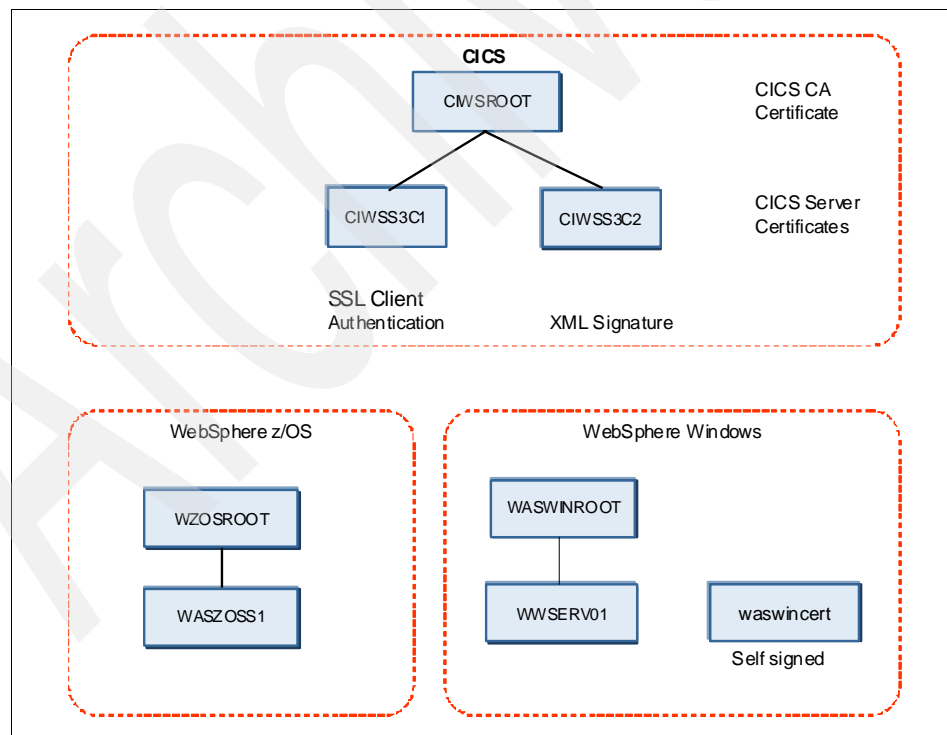


Figure 5-6 Certificates used in the scenarios

5.4.1 Creating the CICS CA certificate

We create a CICS CA certificate, which will be used to establish the trust between the client and CICS in the “SSL Client Authentication” scenario.

First we use the RACDCERT command to create the certificate in RACF. The command is listed in Example 5-1.

Example 5-1 RACDCERT command to create CICS CA certificate

```
RACDCERT CERTAUTH GENCERT +
SUBJECTSDN(CN('CIWSROOT') T('CIWSROOT-certificate') +
OU('ITSO') +
O('IBM') +
L('Poughkeepsie') +
SP('New York') +
C('US')) +
WITHLABEL('CIWSROOT') +
NOTAFTER(DATE(2010/12/31)) +
SIZE(1024) +
ICSF
```

Next we export the certificate to a dataset, and use the OPUT command to copy the exported certificate to an HFS file, from where it can be imported by the client. The commands we use are listed in Example 5-2.

Example 5-2 Export and copy the certificates to an HFS file

```
RACDCERT CERTAUTH EXPORT(LABEL('CIWSROOT')) +
DSN('CIWS.CIWSROOT.DER') FORMAT(CERTDER)
OPUT 'CIWS.CIWSROOT.DER' '/CIWS/CERTIFICATES/CIWSROOT.CER' +
BINARY CONVERT(NO)
```

5.4.2 Creating the WebSphere CA Certificate

Next we create the WebSphere CA certificate. We use the following commands to make the certificate available for the client to use.

First we use the RACDCERT command to create the certificate in RACF. The command is listed in Example 5-3

Example 5-3 RACDCERT command to create WebSphere CA certificate

```
RACDCERT CERTAUTH GENCERT +
SUBJECTSDN(CN('WASWINROOT')) +
```

```
T('WASWINROOT-certificate') +  
OU('ITSO') +  
O('IBM') +  
L('Montpellier') +  
SP('Herault') +  
C('France')) +  
WITHLABEL('WASWINROOT') +  
NOTAFTER(DATE(2010/12/31)) +  
SIZE(1024)
```

Next we export the certificate to a dataset, and use the OPUT command to copy the exported certificate to an HFS file, from where it can be imported by the client. The commands we use are listed in Example 5-4.

Example 5-4 Export and copy certificate to an HFS file

```
RACDCERT CERTAUTH EXPORT(LABEL('WASWINROOT')) +  
DSN('CIWS.WASWROOT.DER') FORMAT(CERTDER) +  
OPUT 'CIWS.WASWROOT.DER' '/CIWS/certificates/WASWINROOT.CER' +  
BINARY CONVERT(NO)  
ICSF
```

5.4.3 Building the key ring

In CICS, the required server certificate and related information about certificate authorities are held in a key ring in the RACF database. The key ring contains the system's private and public key pair, together with the server certificate and the certificates for all the certificate authorities that might have signed the certificates you receive from your clients.

The sample clist DFH\$RING is provided by CICS TS V3.2 to help you build a RACF key ring for CICS SSL use.

We use the command listed in Example 5-5 to invoke the clist and to build the key ring:

Example 5-5 Building a keyring

```
EXEC 'CICSTS32.CICS.SDFHSAMP(DFH$RING)' +  
'CiwS Ciwss3c1 wtsc66.itso.ibm.com FORUSER(CIWS3D)'
```

5.4.4 Defining the CICS resources in RACF

Next we define the transaction IDs and user IDs in RACF, and we permit user access to the pipeline alias transactions.

The commands we use to define the user IDs are listed in Example 5-6.

Example 5-6 ADDUSER command

```
ADDUSER PUBLIC01 NOPASSWORD
ADDUSER PRIVAT01 NOPASSWORD
ADDUSER USERWS01 NOPASSWORD
ADDUSER USERWS02 NOPASSWORD
ADDUSER USERWS03 NOPASSWORD
ADDUSER USERWS04 NOPASSWORD
ADDUSER USERWS05 NOPASSWORD
```

The commands we use to define the transaction IDs are listed in Example 5-7.

Example 5-7 RACF commands to define transactions and transaction groups

```
RDEFINE TCICSTRN CIWS3D.INQS
RDEFINE TCICSTRN CIWS3D.INQC
RDEFINE TCICSTRN CIWS3D.ORDER
RDEFINE TCICSTRN CIWS3D.ORDS
```

The commands we use to permit the pre-defined user IDs to start the CICS pipeline alias transactions are listed in Example 5-8.

Example 5-8 RACF commands to authorize access to pipeline alias transactions

```
PERMIT CIWS3D.INQS CLASS(TCICSTRN) ID(PUBLIC01) ACCESS(READ)
PERMIT CIWS3D.INQC CLASS(TCICSTRN) ID(PUBLIC01) ACCESS(READ)
PERMIT CIWS3D.ORDER CLASS(TCICSTRN) ID(PRIVAT01) ACCESS(READ)
```

Note: The RACF commands that we use to permit end user access to the secure service order transaction ORDS are documented in the scenario chapters.

To activate the definitions, we issue the following command:

```
SETROPTS RACLIST(TCICSTRN) REFRESH
```

5.4.5 Modifying the wsbind file using CICS SupportPac CS04

By default, CICS runs all Web service requests under the same transaction ID CPIH, and the CICS default user ID specified in the SIT.

When you install a PIPELINE resource, CICS scans the directory specified in the PIPELINE's WSDIR attribute (the pickup directory) for wsbind files, and creates URIMAP and WEBSERVICE resources dynamically. Since we want to use the URIMAP definition to specify an alternative transaction ID and user ID, and we are using existing wsbind files, we use CICS SupportPac CS04 *CICS TS for z/OS: WSBIND File Display and Change Utility* to change the wsbind files for the Catalog application services.

Using SupportPac CS04, the following content of the wsbind file can be changed:

TRANSACTION=	Used to add or change a transaction name.
user ID=	Used to add or change a user ID.

Creating the HFS directories

In order to modify the CICS supplied wsbind files, we create a set of new HFS directories that we can use as output files for the utility. The directories we create are listed in Example 5-9. We will later use these directories when we configure the PIPELINE in “Configuring the PIPELINE definition” on page 172.

Example 5-9 HFS directories used in the PIPELINE definition

```
/CIWS/S3C1/config  
/CIWS/S3C1/shelf  
/CIWS/S3C1/wsbind/provider
```

Modifying the WSBIND files

We modify the following wsbind files using SupportPac CS04:

- ▶ inquireSingle.wsbind
- ▶ inquireCatalog.wsbind
- ▶ placeOrder.wsbind

We specify /CIWS/S3C1/wsbind/provider as the output directory and the CICS supplied directory /usr/lpp/cicsts/cicsts32/samples/webservices/wsbind/provider/ as the input directory.

The job we use is shown in Example 5-10.

Example 5-10 Job used to modify wsbind files

```
// SET QT='''
//WSBINDUT EXEC CS04PROC,
// TMPFILE=&QT.&SYSUID.&QT.
//INPUT.SYSUT1 DD *
#-> Add a Tranid and user ID to a WSBind file
INPUT=/usr/lpp/cicsts/cicsts32/samples/webservices/wsbind/provider/*
inquireCatalog
OUTPUT=/CIWS/S3C1/wsbind/provider/inquireCatalog
TRANSACTION=INQC
user ID=PUBLIC01
#-> Add a Tranid to a WSBind file and overwrite
INPUT=/usr/lpp/cicsts/cicsts32/samples/webservices/wsbind/provider/*
inquireSingle
OUTPUT=/CIWS/S3C1/wsbind/provider/inquireSingle
TRANSACTION=INQS
user ID=PUBLIC01
#-> Add a Tranid to a WSBind file and overwrite
INPUT=/usr/lpp/cicsts/cicsts32/samples/webservices/wsbind/provider/*
placeOrder
OUTPUT=/CIWS/S3C1/wsbind/provider/placeOrder
TRANSACTION=ORDR
user ID=PRIVAT01
/*
```

A sample output from the utility is shown in Example 5-11.

Example 5-11 Sample output from SupportPac CS04

```
==> WSBind File Display and Change Utility starting (V1.0.1)
==> Collecting statements. Statements will be scanned twice. If there are any
detected problems on the first pass,
    no actions will be taken.
==> -> stmt 1, comment => #-> Add a Tranid to a WSBind file and overwrite
==> -> stmt 2, =====>
INPUT=/usr/lpp/cicsts/cicsts32/samples/webservices/wsbind/provider/placeOrder
==> -> stmt 3, =====> OUTPUT=/CIWS/S3C6/wsbind/provider/placeOrder
==> -> stmt 4, =====> TRANSACTION=ORDR
==> -> stmt 5, =====> user ID=PRIVAT01

==> ==> Starting pass 1 (a syntax check).

==> ==> Starting pass 2 (taking actions).
```

```
==> -> statement 2 =>
INPUT=/usr/lpp/cicsts/cicsts32/samples/webservices/wsbind/provider/placeOrder
==> Input WSBind file name set to
/usr/lpp/cicsts/cicsts32/samples/webservices/wsbind/provider/placeOrder.wsbind
Details for
/usr/lpp/cicsts/cicsts32/samples/webservices/wsbind/provider/placeOrder.wsbind
- Creation Timestamp: 20050914 08:23
- XML Conversion: Interpretive XML Conversion
- Mapping Level: 1.0
- Minimum Runtime Level: 1.0
- Provider
- Target program name: DFHOXCMN
- Program interface: COMMAREA
- Transaction ID: -none specified-
- User ID: -none specified-
- URI: /exampleApp/placeOrder
- WSDL File Name: /u/chrisb/WasV6/wsd1/placeOrder.wsd1

- Operation: DFHOXCMNOperation
- WSDL Binding: DFHOXCMNHTTPSBinding

==> -> statement 3 => OUTPUT=/CIWS/S3C6/wsbind/provider/placeOrder
==> Output WSBind file name set to /CIWS/S3C6/wsbind/provider/placeOrder.wsbind
===> WSBind File /CIWS/S3C6/wsbind/provider/placeOrder.wsbind already exists,
but will be overwritten...
===> WSBind File was written to /CIWS/S3C6/wsbind/provider/placeOrder.wsbind

==> -> statement 4 => TRANSACTION=ORDR
==> => The old value for Transaction Id was "null"
==> => The new value for Transaction Id is "ORDR"

==> -> statement 5 => user ID=PRIVAT01
==> => The old value for User Id was "null"
==> => The new value for User Id is "PRIVAT01"

==> Output WSBind file name set to placeOrder.wsbind
===> WSBind File placeOrder.wsbind already exists, but will be overwritten...
===> WSBind File was written to placeOrder.wsbind

==> WSBind File Display and Change Utility ending
```

5.4.6 Creating the CICS resource definitions

In this section we describe the CICS resource definitions that are required for the basic configuration.

Configuring the TCPIP SERVICE definition

We define the TCPIP SERVICE resource definition using the following CICS CEDA command:

```
CEDA DEFINE TCPIP SERVICE(S3C1) GROUP(S3C1)
```

We define the S3C1 TCPIP SERVICE as shown Figure 5-7.

```
OVERTYPE TO MODIFY                                     CICS RELEASE = 0650
CEDA DEFine TCpipservice( S3C1      )
TCpipservice      : S3C1
GRoup             : S3C1
DEscription      ==> TCPIP SERVICE DEFINITION FOR CATALOG APPLICATION
Urm              ==> NONE
PORtnumber       ==> 14301          1-65535
STatus           ==> Open          Open | Closed
PROtocol         ==> Http          Iiop | Http | Eci | User
TRansaction      ==> CWXN
Backlog          ==> 00001         0-32767
TSqprefix        ==>
Ippaddress       ==>
SOcketclose      ==> No           No | 0-240000 (HHMSS)
Maxdatalen       ==> 000032       3-524288
SECURITY
SS1              ==> No           Yes | No | Clientauth
CErtificate      ==>
(Mixed Case)

SYSID=S3C1 APPLID=A6POS3C1
```

Figure 5-7 CEDA DEFINE TCPIP SERVICE

We set the PORTNUMBER to 14301, the PROTOCOL to HTTP, and the URM to NONE. We allow the other attributes to default, and we install the S3C1 group.

Customizing the pipeline configuration file

We want the placeOrder service to run initially with transaction ID ORDR and under a fixed user ID PRIVAT01. We then want to switch the security context for this service so that the target business logic program runs under the Web service requester's identity and CICS starts a new task with transaction ID ORDS.

To enable this transaction ID switch, we write a message handler program **CIWSMSGO** which replaces the transaction ID in the DFHWS-TRANID container with the transaction ID **ORDS** when the service request (which can be retrieved from the DFHWS-WEBSERVICE container) is a placeOrder.

To activate the message handler program, we change the PIPELINE configuration file. We copy the file, basicsoap12provider.xml to the /CIWS/S3C1/config directory and make the changes shown in Example 5-12.

Example 5-12 ITSO_7658_basicsoap12provider.xml, pipeline configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline
xmlns="http://www.ibm.com/software/htp/cics/pipeline"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
provider.xsd ">
  <transport>
    <default_transport_handler_list>
    </default_transport_handler_list>
  </transport>
  <service>
    <service_handler_list>
      <handler>
        <program>CIWSMSGO</program>
        <handler_parameter_list/>
      </handler>
    </service_handler_list>
    <terminal_handler>
      <cics_soap_1.2_handler/>
    </terminal_handler>
  </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

The message handler program: CIWSMSGO

The full program listing of CIWSMSGO is shown in Appendix C, “Sample message handler” on page 379.

The following sequence of examples show the logic flow for the message handler.

Example 5-13 shows the overall flow of control. The message handler is invoked for all Web services that are associated with this pipeline.

Example 5-13 CIWSMSGO - evaluate function

```
IF WS-DFHFUNCTION equal 'RECEIVE-REQUEST'  
    PERFORM VALIDATE-REQUEST THRU END-VAL-REQUEST  
    PERFORM CHANGE-TRANID    THRU END-CHANGE-TRANID  
    EXEC CICS  
        DELETE CONTAINER('DFHRESPONSE')  
    END-EXEC  
END-IF  
EXEC CICS RETURN END-EXEC.  
GOBACK.
```

Example 5-14 shows the code to get the Web service request from the DFHWS-WEBSERVICE container.

Example 5-14 CIWSMSGO - get WEBSERVICE container

```
GET-SOAP-WEBSERVICE.  
EXEC CICS  
GET CONTAINER('DFHWS-WEBSERVICE')  
SET(ADDRESS OF CONTAINER-DATA)  
FLENGTH(CONTAINER-LEN)  
END-EXEC.
```

Example 5-15 shows the code that determines the new transaction ID, replacing the transaction ID in the DFHWS-TRANID container with ORDS if the Web service request is placeOrder.

Example 5-15 CIWSMSGO - determine new transaction id

```
CHANGE-TRANID.  
EXEC CICS GET CONTAINER('DFHWS-TRANID')  
SET(ADDRESS OF CONTAINER-DATA)  
FLENGTH(CONTAINER-LEN)  
END-EXEC.  
IF WS-WEBSERVICES = 'placeOrder'  
MOVE 'ORDS' TO CA-TRANID  
PERFORM CHANGE-CONTAINER THRU END-CHANGE-CONTAINER  
END-IF.  
END-CHANGE-TRANID. EXIT.
```

Example 5-16 shows how the program changes the transaction ID in the DFHWS-TRANID container, and performs an EXEC CICS PUT CONTAINER.

Example 5-16 CIWSMSGO - change transaction ID

```
CHANGE-CONTAINER.  
  MOVE CA-TRANID TO CONTAINER-DATA(1:4)  
  EXEC CICS PUT CONTAINER('DFHWS-TRANID')  
    FROM(CONTAINER-DATA)  
    LENGTH(CONTAINER-LEN)  
END-EXEC.
```

Configuring the PIPELINE definition

Next we define the PIPELINE for the CICS service provider using the following CICS CEDA command:

```
CEDA DEFINE PIPELINE(EXPIPE01) GROUP(S3C1)
```

We define the EXPIPE01 pipeline as shown in Figure 5-8.

```
OVERTYPE TO MODIFY                                     CICS RELEASE = 0650  
CEDA DEFINE PIPELINE(EXPIPE01 )  
  Pipeline      : EXPIPE01  
  Group         : S3C1  
  Description   ==>  
  Status        ==> Enabled           Enabled | Disabled  
  Configfile    ==> /CIWS/S3C1/config/ITS0_7658_basicsoap12provider.xml  
  (Mixed Case) ==>  
  ==>  
  ==>  
  ==>  
  SHelf        ==> /CIWS/S3C1/shelf  
  (Mixed Case) ==>  
  ==>  
  ==>  
  ==>  
  Wsdir        : /CIWS/S3C1/wsbind/provider/  
  (Mixed Case) :  
  :
```

Figure 5-8 CEDA DEFINE PIPELINE

- ▶ We set CONFIGFILE to the name of our pipeline configuration file:
/CIWS/S3C1/config/ITSO_7658_basicsoap12provider.xml
- ▶ We set SHELF to the name of the shelf directory:
/CIWS/S3C1/shelf
- ▶ We set WSDIR to the Web service binding directory that contains the wsbind files for the sample application:
/CIWS/S3C1/wsbind/provider/

Installing the PIPELINE definition

Next we use CEDA to install the PIPELINE definition. When the PIPELINE is installed CICS scans the wsdir directory, and dynamically creates WEBSERVICE and URIMAP definitions for the wsbind files found.

Figure 5-9 shows a CEMT INQUIRE PIPELINE for EXPIPE01.

```
INQUIRE PIPELINE
RESULT - OVERTYPE TO MODIFY
  Pipeline(EXPIPE01)
  Enablestatus( Enabled )
  Configfile(/CIWS/S3C1/config/ITSO_7658_basicsoap12provider.xml)
  Shelf(/CIWS/S3C1/shelf/)
  Wsdir(/CIWS/S3C1/wsbind/provider/)

SYSID=S3C1 APPLID=A6POS3C1
```

Figure 5-9 CEMT INQUIRE PIPELINE - EXPIPE01

Defining the transactions

We copy the CPIH transaction definition to create the CICS transaction IDs for the Catalog service requests, as listed in Example 5-17.

Example 5-17 CICS definitions - TRANSACTION

```
CEDA COPY TRANSACTION(CPIH) GROUP(DFHPIPE) TO(S3C1) AS(INQS)
CEDA COPY TRANSACTION(CPIH) GROUP(DFHPIPE) TO(S3C1) AS(INQC)
CEDA COPY TRANSACTION(CPIH) GROUP(DFHPIPE) TO(S3C1) AS(ORDR)
CEDA COPY TRANSACTION(CPIH) GROUP(DFHPIPE) TO(S3C1) AS(ORDS)
```

5.5 The CICS catalog manager example application

The CICS catalog example application is a COBOL application that is designed to illustrate best practice when connecting CICS applications to external clients and servers. The example is constructed around a simple sales catalog and order processing application, in which the end user can perform these functions:

- ▶ List the items in a catalog.
- ▶ Inquire on individual items in the catalog.
- ▶ Order items from the catalog.

The catalog is implemented as a VSAM file. The base application has a 3270 user interface, but the modular structure, with well-defined interfaces between the components, makes it possible to add further components. In particular, the application comes with Web service support, which is designed to illustrate how you can extend an existing application into the Web services environment.

5.5.1 Installing and setting up the application

Before we run the application, we define and populate two KSDS VSAM datasets, and define two transactions.

Creating and defining the VSAM datasets

To create the datasets two set of JCL are provided by CICS. These are located in CICSTS32.CICS.SDFHINST and are named:

DFH\$ECFN JCL to generate the configuration dataset.
DFHECAT JCL to generate catalog dataset

Defining the 3270 interface

The example application is supplied with a 3270 user interface to run the application and to customize it. The user interface consists of two transactions, EGUI and ECFG. We use the CEDA commands shown in Example 5-18 to define and install the transactions.

Example 5-18 Defining the Catalog application transactions.

```
CEDA DEFINE TRANSACTION(EGUI) PROGRAM(DFHOXGUI) GROUP(S3C1)
CEDA DEFINE TRANSACTION(ECFG) PROGRAM(DFHOXCFG) GROUP(S3C1)
CEDA INSTALL GROUP(S3C1)
```

Note: Because we use program autoinstall, we do not define the programs and BMS maps used by the application. For a full list of resources used, see the CICS TS 3.2 Information Center, at the following URL:

<http://publib.boulder.ibm.com/infocenter/cicsts/v3r2/index.jsp>

Installing the catalog application Web client

The Web service support extends the example application, providing a Web client front end. The client is delivered in two versions, which will run in the environments shown in Table 5-6. In our environment, we use ExampleAppClientV6.ear.

Table 5-6 The Web clients and environments.

Environment	EAR files
WebSphere Application Server Ver. 5.1	ExampleAppClient.ear
WebSphere Application Server Ver. 6.1	ExampleAppClientV6.ear

The CICS supplied ear files are located in the HFS directory:

`/usr/lpp/cicsts32/samples/webservices/client/`

We install the Web client on WebSphere Application Server V6.1 running on Windows.

5.5.2 Running the application

To configure the Web client application, we enter the following URL:

<http://Cev8-Pc6:9080/ExampleAppClientV6Web>

On the window presented in Figure 5-10, we enter the URL of our Web service provider CICS region. The hostname is set to **wtsc66.itso.ibm.com** and the port is set to **14301**.

Inquire Catalog Service Endpoint	
Current	http://wtsc66.itso.ibm.com:14301/exampleApp/inquireCatalog
New	http://wtsc66.itso.ibm.com:14301/exampleApp/inquireCatalog

Inquire Item Service Endpoint	
Current	http://wtsc66.itso.ibm.com:14301/exampleApp/inquireSingle
New	http://wtsc66.itso.ibm.com:14301/exampleApp/inquireSingle

Place Order Service Endpoint	
Current	http://wtsc66.itso.ibm.com:14301/exampleApp/placeOrder
New	http://wtsc66.itso.ibm.com:14301/exampleApp/placeOrder

Figure 5-10 Configuring the CICS catalog manager example application

We click **SUBMIT** to submit our configuration changes.

Next we run the inquireSingle Web service, the result of which is shown in Figure 5-11.

The screenshot shows a web application interface titled "CICS Example - Catalog Application". The main heading is "Item Details - Select to Place Order". On the left side, there are three buttons: "LIST ITEMS", "INQUIRE", and "ORDER ITEM". The main content area is a table with the following data:

Item	Description	In Stock	On Order	Cost	Select
0010	Ball Pens Black 24pk	2493	2	\$2.90	6

At the bottom right of the table area, there is a "SUBMIT" button.

Figure 5-11 inquireSingle Web services window

While the request is active in CICS, we enter a CEMT INQUIRE TASK to verify that the Web service request inquireSingle runs under the correct user ID and transaction ID (see Figure 5-12).

```

I TASK (531)
STATUS: RESULTS - OVERTYPE TO MODIFY
Tas(0000531) Tra(INQS) Sus Tas Pri( 001 )
Sta(U ) Use(PUBLIC01) Uow(C2B97193C05F4F0C) Hty(EDF )

SYSID=S3C1 APPLID=A6POS3C1

```

Figure 5-12 CEMT INQUIRE TASK

Next we click **SUBMIT** to invoke the placeOrder Web service, we fill in the window shown in Figure 5-13, and click **SUBMIT**.

CICS Example - Catalog Application	
Enter Order Details	
LIST ITEMS	Item Reference Number 0010
INQUIRE	Quantity 001
	User Name Tommy
	Department Name ITSO
	SUBMIT

Figure 5-13 The Enter Order Details window

In Example 5-19 we show the RACF messages that are issued. The message is issued because user PRIVAT01 is not allowed to attach transaction ORDS.

Example 5-19 RACF messages from attempt to run ORDS without permission

```
ICH408I USER(PRIVAT01) GROUP(SYS1 ) NAME(PRIVAT01 FOR ORDR )
      CIWS3D.ORDS CL(TCICSTRN)
      INSUFFICIENT ACCESS AUTHORITY
      ACCESS INTENT(READ ) ACCESS ALLOWED(NONE )
```

In Example 5-20 we show the messages from the CICS log. We see that user PRIVAT01 is not allowed to run the ORDS transaction, and therefore the initial transaction ORDR fails.

Example 5-20 CICS error messages

```
DFHXS1111 07/16/2008 04:29:19 A6POS3C1 ORDS Security violation by user
PRIVAT01 for resource CIWS3D.ORDS in class TCICSTRN. SAF
codes are (X'00000008',X'00000000'). ESM codes are
(X'00000008',X'00000000').
DFHAC2003 07/16/2008 04:29:19 A6POS3C1 Security violation has been
detected term id = ????, trans id = ORDS, user ID = PRIVAT01.
DFHPI0507 07/16/2008 04:29:19 A6POS3C1 ORDR The CICS Pipeline Manager
has failed to receive a response from an application handling
task.The connection to the application task was closed.
PIPELINE: EXPIPE01.
DFHPI0997 07/16/2008 04:29:19 A6POS3C1 ORDR EXPIPE01 The CICS pipeline
manager has encountered an error: unidentified problem.
```

In the following chapters we describe the different security scenarios that we configured. In these scenarios, we configure WebSphere Application Server and CICS such that a specific identity is passed to CICS. The RACF ID associated with this identity is then authorized to run the placeOrder service.

Archived

Enabling SSL

By default, SOAP messages are flowed in clear text. This is likely to be unacceptable in many cases. In order to address this, some form of protection is required.

SSL/TLS is a well understood and popular way of providing encryption and data integrity. When the client connects with SSL/TLS, privacy of the data is obtained by encrypting the data, and data integrity is achieved using a digital signature. In addition to privacy and data integrity, we can also use SSL/TLS for authentication by the use of a client certificate.

In this section we show how we configured both the CICS and WebSphere environments to use an SSL/TLS client authenticated connection. We provide step-by-step security configurations for configuring the CICS service provider pipeline and the WebSphere service requester using the WebSphere Administration Console.

6.1 Preparation

In this section we provide a summary of the software used in the scenario and the main definitions used for the CICS and WebSphere servers.

6.1.1 Software checklist

The software we used is listed in Table 6-1.

Table 6-1 Software used in the XML digital signature scenario

Windows	z/OS
Internet Explorer® V7.0	z/OS V1.9
Windows XP Professional Version 2002 SP3	CICS Transaction Server V3.2
IBM WebSphere Application Server V6.1.0.17	
IBM WebSphere Application Server Toolkit V6.1.1.6	
Our J2EE application <ul style="list-style-type: none">▶ ExampleAppClientV6.ear The Web client for the CICS catalog example application supplied with CICS TS.	

6.1.2 Definition checklist

The definitions we used are listed in Table 6-2.

Table 6-2 Settings used in the SSL scenario

Value	CICS TS	WebSphere Application Server
IP name	wtsc66.itso.ibm.com	cev8-pc12
TCP/IP ports	14301 and 14311	9080
Jobname	CIWSS3C1	
APPLID	A6POS3C1	
TCPIPSERVICE	S3C1 (port 14301 HTTP) and S3C1SSL (port 14311 HTTPS)	
Provider PIPELINES	EXPIPE01 for inquireSingle, inquireCatalog services and placeOrder services	

Value	CICS TS	WebSphere Application Server
Configuration files	ITSO_7658_basicoap12provider.xml for inquireSingle, inquireCatalog and placeOrder services	

The certificates we used in our configuration are listed in Table 6-3.

Table 6-3 Certificates used in the SSL scenario

Value	Format	File
CICS server certificate	PKCS12DER	CIWSS3C1.P12
CICS CA certificate	CERTDER	CIWSROOT.CER
WebSphere client certificate	CER	waswincert.cer

The user IDs we used in our configuration are listed in Table 6-4.

Table 6-4 User IDs used in the SSL scenario

Value	CICS TS
CICS region user ID	CIWS3D
User ID for which we want to permit access to inquireSingle and inquireCatalog services	PUBLIC01
User ID for which we want to permit access to the ORDS transaction of the placeOrder service	USERWS02

Because the Catalog application inquireSingle and inquireCatalog services are read-only services, whereas the placeOrder service updates the database, we show here how to secure the placeOrder service.

We run the inquireSingle and inquireCatalog services under a predefined user ID PUBLIC01, which is defined in the inquireSingle and inquireCatalog wsbind files.

6.1.3 Scenario overview

To provide integrity, privacy, and authentication between the Web Service requester and the Web Service provider, we will use HTTP over Secure Sockets Layer, HTTPS, as our transport. This will provide transport level security for the placeOrder Web Service. The inquireSingle and inquireCatalog Web Services do not require security and will use unsecured HTTP.

3. WebSphere receives the CICS server certificate, and verifies that the details on the certificate match the origin of the certificate, for example the hostname. It then checks in its trust store looking for the CICS CA certificate. If it finds a match, then the CICS server is trusted.
4. The SOAP message is sent over HTTPS. The encryption and signature are enabled at the transport level.
5. The message arrives in CICS, and CICS finds a URIMAP that matches the incoming request. From the URIMAP, CICS determines the Web Service (placeOrder), the Pipeline (EXPIPE01), the Transaction (ORDS) and the user ID (PRIVAT01).
6. User ID PRIVAT01 does not have the correct RACF authority to run transaction ORDS. CICS returns an error to WebSphere Application Server in a SOAP fault message.

The SSL client authentication scenario is shown in Figure 6-2.

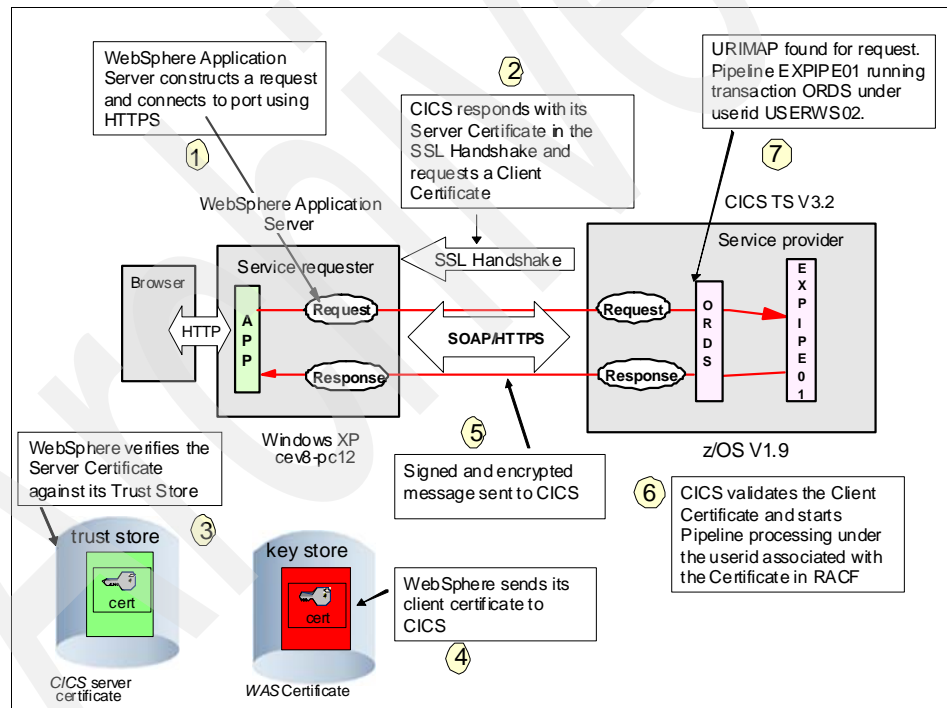


Figure 6-2 SSL client authentication scenario

The sequence of steps is as follows:

1. WebSphere Application Server constructs a request and then initiates the connection to CICS over HTTPS.

2. CICS responds with its server certificate.
3. WebSphere receives the CICS server certificate, and verifies that the details on the certificate match the origin of the certificate, for example the hostname. It then checks in its trust store looking for the CICS CA certificate. If it finds a match, then the CICS server is trusted.
4. WebSphere extracts its client certificate from the key store and sends it to CICS.
5. The SOAP message is sent over HTTPS. The encryption and signature are done at the transport level.
6. CICS validates the client certificate using the WebSphere CA certificate which is stored in the CICS key ring. It also maps the client certificate to a RACF user ID USERWS02.
7. CICS finds a URIMAP which matches the incoming request. From the URIMAP, CICS determines the Web Service (placeOrder), the Pipeline (EXPIPE01), the Transaction (ORDS) and the user ID (PRIVAT01).

The ORDS transaction is attached using the USERWS02 user ID.

Note: A user ID associated with a client certificate takes precedence over a user ID specified on a URIMAP definition.

User ID USERWS02 has the correct RACF authority to run transaction ORDS. The Web Service completes and a normal response message is returned to WebSphere.

In this chapter, we document the following procedures to enable support for server-side SSL processing:

- ▶ Creating the CICS server certificate
- ▶ Configuring the WebSphere service requester for server-side SSL processing
- ▶ Configuring the CICS service provider for server-side SSL processing
- ▶ Deploying the Web service application
- ▶ Testing the server-side SSL scenario

We then document the additional procedures to enable support for SSL client authentication:

- ▶ Creating the WebSphere client certificate
- ▶ Configuring the WebSphere service requester for client certificate SSL processing
- ▶ Configuring the CICS service provider for client certificate SSL processing

- ▶ Testing SSL client authentication

6.2 Creating the CICS server certificate

In order to enable server-side SSL processing, we create a certificate and key pair to be used by CICS, and then connect the certificate to the CICS key ring.

Example 6-1 shows the RACF command that was used to create the CICS certificate.

Example 6-1 CICS certificate (CIWSS3C1)

```
RACDCERT ID (CIWS3D) GENCERT
SUBJECTSDN(CN('CIWSS3C1')
T('CIWSS3C1-certificate')
OU('ITSO')
O('IBM')
L('Poughkeepsie')
SP('New York')
C('US'))
WITHLABEL('CIWSS3C1')
SIGNWITH(CERTAUTH LABEL('CIWSROOT'))
KEYUSAGE(HANDSHAKE DATAENCRYPT DOCSIGN)
NOTAFTER(2010/12/31)
SIZE(1024)
PASSWORD('PASSWORD')
```

In Example 6-1:

1. The RACDCERT GENCERT command creates the certificate and a public/private key pair.
2. ID specifies that the CICS region user ID CIWS3D is to be associated with the certificate.
3. SUBJECTSDN specifies the subject's X.509 distinguished name.
4. WITHLABEL specifies the label name to be associated with the certificate.
5. SIGNWITH specifies the certificate with a private key that is signing the certificate. The CICS certificate is signed with the CICS CA certificate that was created in Chapter 5-1, "Enabling SSL" on page 153.

6. KEYUSAGE specifies how the keys associated with the certificate are to be used. We specify:
 - HANDSHAKE because the certificate will be used for SSL handshakes
 - DATAENCRYPT because the certificate will be used for encryption
 - DOCSIGN because the certificate will be used for signing
- ▶ NOTAFTER specifies the date after which the certificate is no longer valid.
- ▶ SIZE specifies the size of the private key expressed in decimal bits. We specified a high strength key length of 1024.

Note: A certificate that CICS uses for SSL processing does not have to be created with the ICSF option. Since the Crypto Express 2 Accelerator (CEX2A) only works with clear keys, there might be a performance advantage in not using the ICSF option.

Example 6-2 shows the RACF commands that we used to connect the CICS X.509 certificate to the key ring and to list the certificate.

Example 6-2 RACF command to connect CICS certificate to RACF key ring

```
RACDCERT ID(CIWS3D) CONNECT(ID(CIWS3D) LABEL('CIWSS3C1') RING(Ciws.Ciwss3c1))
RACDCERT ID(CIWS3D) LIST
```

Example 6-3 shows the output of the RACDCERT LIST command.

Example 6-3 Listing of CICS certificate (CIWSS3C1)

Digital certificate information for user CIWS3D:

```
Label: CIWSS3C1
Certificate ID: 2QbDyebi88TDyebi4vPD8UBA
Status: TRUST
Start Date: 2008/07/22 00:00:00
End Date: 2010/12/31 23:59:59
Serial Number:
    >0C<
Issuer's Name:
    >CN=CIWSROOT.T=CIWSROOT-certificate.OU=ITS0.0=IBM.L=Poughkeepsie.SP=Ne<
    >w York.C=US<
Subject's Name:
    >CN=CIWSS3C1.T=CIWSS3C1-certificate.OU=ITS0.0=IBM.L=Poughkeepsie.SP=Ne<
    >w York.C=US<
```

```
Key Usage: HANDSHAKE, DATAENCRYPT, DOCSIGN
Private Key Type: Non-ICSF
Private Key Size: 1024
Ring Associations:
  Ring Owner: CIWS3D
  Ring:
    >CiwS.Ciwss3c1<
```

6.3 Configuring WebSphere Application Server for SSL processing

In this section we describe the configuration of WebSphere Application Server for server-side SSL processing.

6.3.1 Adding the CICS CA certificate to the trust store

In order to configure WebSphere to use server-side SSL, we downloaded the CICS CA certificate from the host and added the certificate to the trust store using the WebSphere Administration Console.

Example 6-23 shows how we used FTP to send the exported certificate (in binary format) to our WebSphere machine.

Example 6-4 FTP WebSphere certificate to WebSphere machine

```
C:\>ftp wtsc66.itso.ibm.com
Connected to wtsc66.itso.ibm.com.
User (wtsc66.itso.ibm.com:(none)): CICSRS6
331-Password:
230-220-FTPMVS1 IBM FTP CS V1R9 at wtsc66.itso.ibm.com, 08:17:08 on 2008-07-08.
230-CICSRS8 is logged on. Working directory is "CICSRS8.".
ftp> cd /CIWS/certificates
250 HFS directory /CIWS/certificates is the current working directory.
ftp> bin
200 Representation type is Image
ftp> get CIWSROOT.CER
200 Port request OK.
125 Sending data set /CIWS/certificates/CIWSROOT.CER
250 Transfer completed successfully.
ftp: 2712 bytes received in 0.00Seconds 2712000.00Kbytes/sec.
ftp> bye
221 Quit command received. Goodbye.
```

To complete the setup of certificates, we now add the CICS CA certificate to the WebSphere trust store. WebSphere uses this CA certificate to validate the CICS certificate which is used to sign the message response.

We use the default WebSphere trust store `NodeDefaultTrustStore`. The default trust store references file `trust.p12` in the node or cell directories of the configuration repository. This file is a PKCS12 type trust store.

Table 6-5 shows the main settings for the trust store and the WebSphere certificate.

Table 6-5 Client trust store and certificate values

Field	Value
trust store name	<code>NodeDefaultTrustStore</code>
Filename	<code>trust.p12</code>
Storepass	<code>WebAS</code>
Storetype	<code>PKCS12</code>
Distinguished Name	<code>CN=CIWSROOT, OU=ITSO, O=IBM</code>
Key size	<code>1024</code>
Alias	<code>cics ca certificate</code>

We added the certificate to the trust store using the WebSphere Administration Console.

We perform the following steps using the WebSphere admin console:

1. Select **Security** → **SSL certificate and key management**.
2. Under Related Items, select **Key stores and certificates**.
3. The resulting panel shows the default key stores and trust stores. Click **NodeDefaultTrustStore**.
4. Under Additional Properties, select **Signer certificates**, click **Add** and enter these values:

Alias: CICS CA certificate
File name: c:/CIWSROOT.CER
Data type: Binary DER file

Figure 6-3 shows the CICS CA certificate being added to the trust store.



Figure 6-3 Adding CICS CA certificate to trust store

5. Click **OK** and save the changes.

Figure 6-4 shows the CICS CA certificate in the NodeDefaultTrustStore.

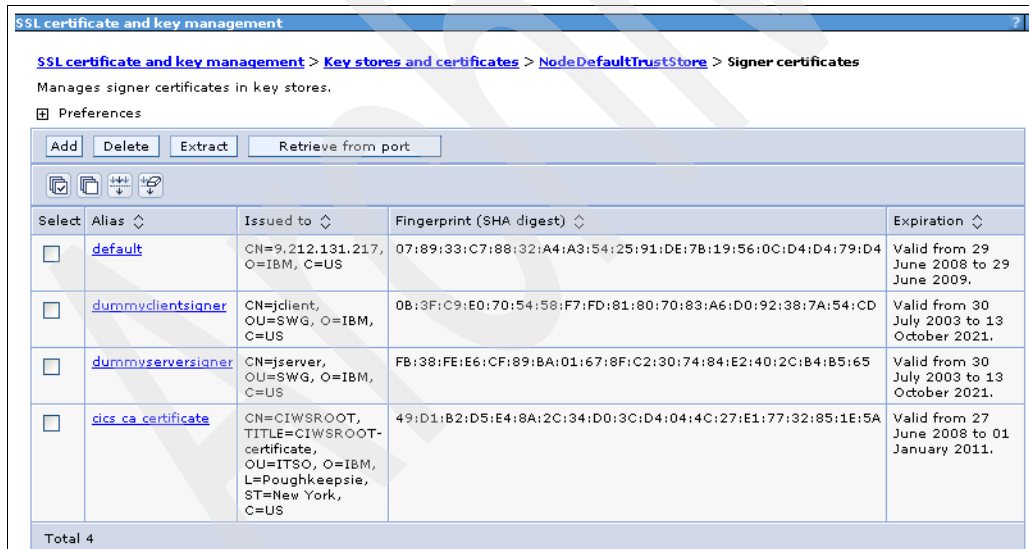


Figure 6-4 Signer certificates in trust store

6.3.2 Configuring WebSphere Application Server's SSL settings

WebSphere Application Server is installed with default SSL settings for the entire cell (for Base installations) or node (for Network Deployment installations). These settings are configurable using the admin console. Further granularity can be obtained by setting up and configuring SSL settings individual nodes, cells and end-points.

For this scenario, we use the default WebSphere NodeDefaultSSLSettings SSL configuration.

We perform the following steps using the WebSphere admin console:

1. Select **Security** → **SSL certificate and key management**.
2. Under **Related Items** select **SSL configurations**.
3. The resulting panel shows the default SSL configuration. Click the **NodeDefaultSSLStore**. The results can be seen in Figure 6-5.



Figure 6-5 SSL configurations in WebSphere Application Server admin console

4. Figure 6-6 shows the settings for the default SSL configuration.

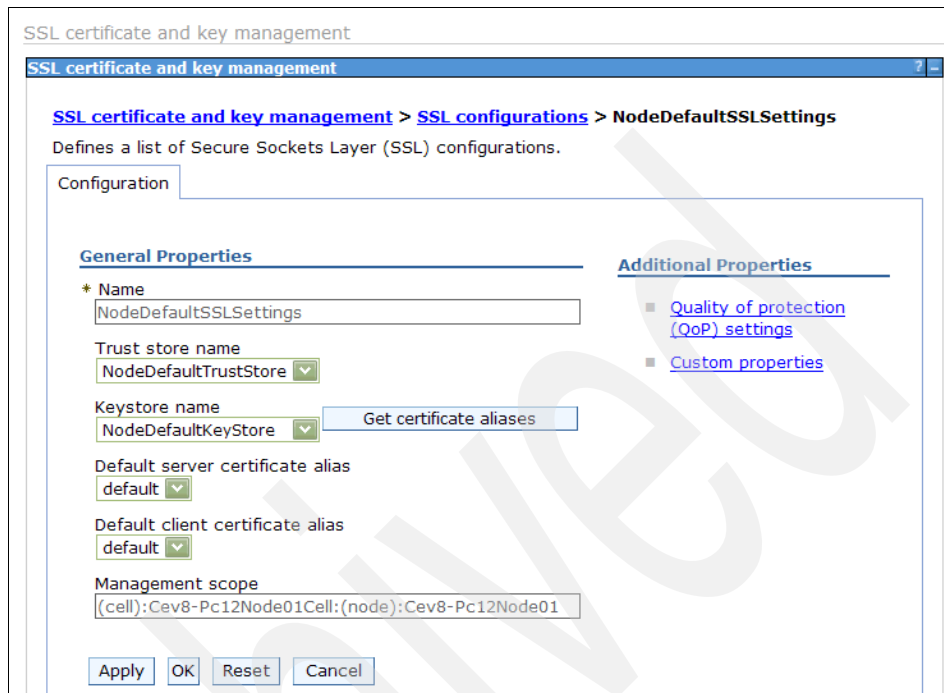


Figure 6-6 Default settings for the default SSL configuration

We allow the settings to default as follows:

- ▶ **Trust store name** is NodeDefaultTrustStore.
 - ▶ **Key store name** is NodeDefaultKeyStore.
 - ▶ **Default server certificate alias** is default.
 - ▶ **Default client certificate alias** is default.
 - ▶ **Management scope** is (cell):Cev8-Pc12Node01Cell:(node):Cev8-Pc12Node01. This uniquely identifies this instance of WebSphere Application Server.
5. Under **Additional Properties**, click **Quality of protection (QoP) settings**. The results can be seen in Figure 6-7.

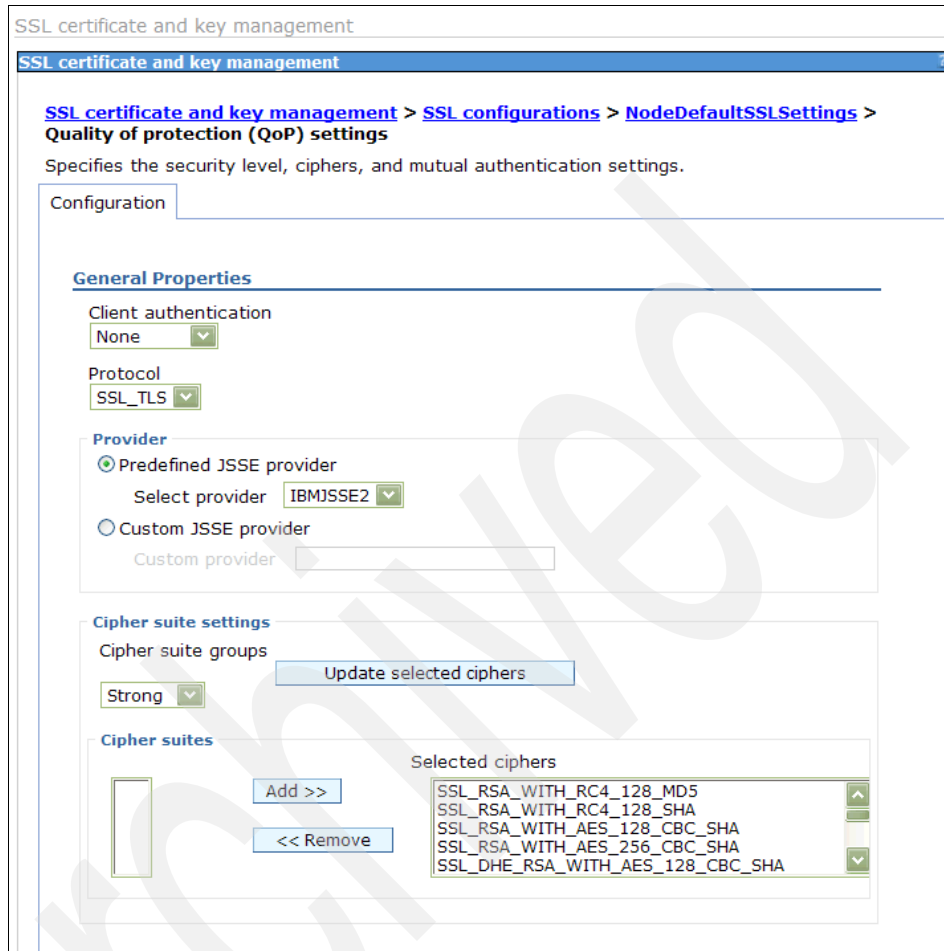


Figure 6-7 Quality of Protection (QoP) settings under SSL configurations

- ▶ **Client authentication:** Applies when WebSphere is acting as an SSL server. This can be set to None, Supported, or Required and relates to whether a Client certificate is required to communicate with WebSphere.
- ▶ **Protocol:** Specifies the SSL handshake protocol. The default is SSL_TLS, which supports all handshake protocols except for SSLv2. WebSphere allows you to specify:
 - SSL
 - SSLv2
 - SSLv3
 - TLS
 - SSL_TLS
 - TLSv1

We let this setting default to SSL_TLS.

- ▶ **Provider:** Specifies which program suite will provide the SSL processing capability. It is possible to use a custom provider. We select the default, **Predefined JSSE provider** and the default JSSE provider, IBMJSSE2.
- ▶ **Cipher suite settings:** Specifies which cipher suites are available for SSL processing. WebSphere Application Server has three standard settings but also permits Custom settings, where Cipher suites are chosen from a list.

The three standard settings are:

- **Strong**, which supports the following ciphers:
 - SSL_RSA_WITH_RC4_128_MD5
 - SSL_RSA_WITH_RC4_128_SHA
 - SSL_RSA_WITH_AES_128_CBC_SHA
 - SSL_RSA_WITH_AES_256_CBC_SHA
 - SSL_DHE_RSA_WITH_AES_128_CBC_SHA
 - SSL_DHE_RSA_WITH_AES_256_CBC_SHA
 - SSL_DHE_DSS_WITH_AES_128_CBC_SHA
 - SSL_DHE_DSS_WITH_AES_256_CBC_SHA
 - SSL_RSA_WITH_3DES_EDE_CBC_SHA
 - SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA
 - SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
 - SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
 - SSL_RSA_WITH_DES_CBC_SHA
 - SSL_RSA_FIPS_WITH_DES_CBC_SHA
 - SSL_DHE_RSA_WITH_DES_CBC_SHA
 - SSL_DHE_DSS_WITH_DES_CBC_SHA
 - SSL_RSA_EXPORT_WITH_RC4_40_MD5 (also in **Medium**)
 - SSL_RSA_EXPORT_WITH_DES40_CBC_SHA (also in **Medium**)
 - SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA (also in **Medium**)
 - SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA (also in **Medium**)

- **Medium**, which supports the following ciphers:
 - SSL_RSA_EXPORT_WITH_RC4_40_MD5 (also in **Strong**)
 - SSL_RSA_EXPORT_WITH_DES40_CBC_SHA (also in **Strong**)
 - SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA (also in **Strong**)
 - SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA (also in **Strong**)
 - SSL_DH_anon_EXPORT_WITH_RC4_40_MD5 (also in **Weak**)
 - SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA (also in **Weak**)
- **Weak**, which supports the following ciphers:
 - SSL_DH_anon_EXPORT_WITH_RC4_40_MD5 (also in **Medium**)
 - SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA (also in **Medium**)
 - SSL_DH_anon_WITH_DES_CBC_SHA
 - SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
 - SSL_DH_anon_WITH_RC4_128_MD5
 - SSL_DH_anon_WITH_AES_128_CBC_SHA
 - SSL_DH_anon_WITH_AES_256_CBC_SHA
 - SSL_RSA_WITH_NULL_MD5 (no encryption)
 - SSL_RSA_WITH_NULL_SHA (no encryption)

There are four ciphers in **Strong** that also appear in **Medium**, and two ciphers in **Medium** that also appear in **Weak**. We select the default, **Strong**, to ensure that our data will always be protected.

We accept the default list of ciphers.

6.4 Configuring CICS for SSL processing

In this section we show how to configure CICS for server-side SSL processing. We document the following steps:

- ▶ Enabling CICS to use SSL
- ▶ Defining a new TCPIP SERVICE to support HTTPS
- ▶ Defining a new WEBSERVICE to process placeOrder request messages
- ▶ Defining a new URIMAP to only allow requests over HTTPS to access the Web Service

Attention: we are not defining a new PIPELINE for this scenario. Our security is at the transport level, not the message level. We therefore need a new TCPIP SERVICE but not a new PIPELINE. The placeOrder Web Service will run securely within the same pipeline, EXPIPE01, as the unsecure Web Services inquireSingle and inquireCatalog.

6.4.1 Enabling CICS to use SSL

In this section we outline the steps required to enable CICS to use SSL.

Ensuring that CICS has access to System SSL libraries

The System SSL library, *hlq.SIEALNKE* must be made available to the CICS region in any of:

- ▶ JOBLIB
- ▶ STEPLIB
- ▶ LINKLIST

We checked the Link List by issuing the command in Example 6-5.

Example 6-5 Command to display which libraries are in the Link List

```
/D PROG,LNKLST
```

The results can be seen in Example 6-6.

Example 6-6 Results of Display Program, LNKLIST command

```
CSV470I 05.29.46 LNKLST DISPLAY 032
LNKLST SET LNKLST66 LNKAUTH=LNKLST
ENTRY  APF  VOLUME  DSNAME
  1     A   TOTSYS2  SYS1.SC66.LINKLIB
  2     A   TOTSYS2  SYS1.P101.LINKLIB
  3     A   TOTSYS2  SYS1.WTSCPLX1.LINKLIB
  4     A   Z19RF1   SYS1.SIEALNKE
```

SYS1.SIEALNKE is in the Link List and is therefore available to the CICS region.

Specifying System Initialization Table (SIT) parameters

Six SIT parameters relate to SSL processing. These are:

- ▶ CRLPROFILE
- ▶ ENCRYPTION
- ▶ KEYSRING

- ▶ MAXSSLCBS
- ▶ SSLCACHE
- ▶ SSLDELAY

Specifying the CRLPROFILE SIT parameter

CRLPROFILE specifies the name of the profile that authorizes CICS to access certificate revocation lists (CRL) that are stored in an LDAP server. There is no default. We have not configured an LDAP server for use with Certificate Revocation Lists and so we do not specify this parameter.

Note: Without this SIT parameter, CICS will not check the revocation status of certificates during SSL handshakes. If a certificate should become compromised, that is, if its private key becomes known, then that certificate is revoked by publishing this information on a CRL. Without checking the CRL, it is possible that SSL communications could be compromised.

Specifying the ENCRYPTION SIT parameter

ENCRYPTION specifies the cipher suites that CICS uses for secure TCP/IP connections. When a secure connection is established between a pair of processes, the most secure cipher suite supported by both is used. This SIT parameter can be set to WEAK, MEDIUM, or STRONG, as follows:

- ▶ **WEAK**, which supports the following Cipher Suites at z/OS 1.9:

Cipher Suite	Encryption Algorithm	Key length	Hash algorithm	Key exchange	Certificate
01	None	None	MD5	RSA	RSA
02	None	None	SHA-1	RSA	RSA
03	RC4	40 bits	MD5	RSA	RSA
06	RC2	128 bits	MD5	RSA	RSA

- ▶ **MEDIUM**, which supports all the Cipher Suites of **WEAK** and also:

Cipher Suite	Encryption Algorithm	Key length	Hash algorithm	Key exchange	Certificate
09	DES	56 bits	SHA-1	RSA	RSA

- **STRONG**, which supports all the Cipher Suites of **MEDIUM** and also:

Cipher Suite	Encryption Algorithm	Key length	Hash algorithm	Key exchange	Certificate
04	RC4	128 bits	MD5	RSA	RSA
05	RC4	128 bits	SHA-1	RSA	RSA
0A	Triple DES	168 bits	SHA-1	RSA	RSA
0C	DES	56 bits	SHA-1	fixed Diffie-Hellman	DSS
0D	Triple DES	168 bits	SHA-1	fixed Diffie-Hellman	DSS
0F	DES	56 bits	SHA-1	fixed Diffie-Hellman	RSA
10	Triple DES	168 bits	SHA-1	fixed Diffie-Hellman	RSA
12	DES	56 bits	SHA-1	ephemeral Diffie-Hellman	DSS
13	Triple DES	168 bits	SHA-1	ephemeral Diffie-Hellman	DSS
15	DES	56 bits	SHA-1	ephemeral Diffie-Hellman	RSA
16	Triple DES	168 bits	SHA-1	ephemeral Diffie-Hellman	RSA
2F	AES	128 bits	SHA-1	RSA	RSA
30	AES	128 bits	SHA-1	fixed Diffie-Hellman	DSS
31	AES	128 bits	SHA-1	fixed Diffie-Hellman	RSA
32	AES	128 bits	SHA-1	ephemeral Diffie-Hellman	DSS
33	AES	128 bits	SHA-1	ephemeral Diffie-Hellman	RSA
35	AES	256 bits	SHA-1	RSA	RSA
36	AES	256 bits	SHA-1	ephemeral Diffie-Hellman	DSS

Cipher Suite	Encryption Algorithm	Key length	Hash algorithm	Key exchange	Certificate
37	AES	256 bits	SHA-1	ephemeral Diffie-Hellman	RSA
38	AES	256 bits	SHA-1	ephemeral Diffie-Hellman	DSS
39	AES	256 bits	SHA-1	ephemeral Diffie-Hellman	RSA

We select ENCRYPTION=STRONG to ensure that we have the most choice of cipher suites.

Important: Unlike WebSphere, where selecting Strong Ciphers means only to use Strong Ciphers, ENCRYPTION=STRONG means that CICS can use all Ciphers: Weak, Medium, or Strong. We discuss how to customize the available cipher suites in 6.4.2, “Defining a new TCPIP SERVICE” on page 201.

Specifying the KEYRING SIT parameter

KEYRING specifies the name of a key ring in the RACF database that contains keys and certificates used by CICS. It must be owned by the CICS region user ID. We created a KEYRING in 5.4, “Creating CA certificates” on page 162.

We specify KEYRING=Ciws.Ciwss3c1 for our CICS region

Specifying the MAXSSLTCBS SIT parameter

MAXSSLTCBS specifies the maximum number of S8 TCBs that are available to CICS to process secure sockets layer connections. This value is a number in the range 0 through 999, and has a default value of 8. This number is the maximum number of concurrent connections that can process SSL simultaneously. Unlike previous versions of CICS, S8 TCBs are managed in a pool and are attached and detached as necessary. In addition, also unlike previous releases of CICS, S8 TCBs are only held for the duration of SSL processing, not for the lifetime of the task. Therefore it is possible to have more SSL connections than there are S8 TCBs in the pool.

We specify MAXSSLTCBS=8 for our CICS region, keeping the default.

Specifying the SSLCACHE SIT parameter

SSLCACHE specifies whether CICS should use the local SSL cache in the CICS region, or share the cache across multiple CICS regions by using the coupling

facility. The valid options are **CICS** or **SYSPLEX**. We specify the default value **CICS** because we only have one CICS region in this scenario.

Note: in a larger environment where many CICS regions are sharing work, **SSLCACHE=SYSPLEX** offers performance improvements, as discussed in “System initialization parameters related to SSL” on page 18.

Specifying the SSLDELAY SIT parameter

SSLDELAY specifies the length of time in seconds for which CICS retains session IDs for secure socket connections in a local CICS region. Session IDs are tokens that represent a secure connection between a client and an SSL server. While the session ID is retained by CICS within the SSLDELAY period, CICS can continue to communicate with the client without the significant overhead of an SSL handshake.

Provided the socket between CICS and the client remains open, then no handshake is necessary until the SSLDELAY interval expires. If the socket does close, but the SSLDELAY has not expired, then a null-handshake occurs. This is very much cheaper in CPU terms than a full handshake.

We specify the default value **600** which, at 10 minutes, represents a balance between too frequent handshaking with the resultant CPU costs and too infrequent handshaking with security considerations.

The CICS region is restarted in order to activate the new SIT parameters, as shown in Example 6-7.

Example 6-7 SIT Parameters for SSL (CRLPROFILE not specified)

```
ENCRYPTION=STRONG,  
KEYRING=Ciws.Ciws3c1,  
MAXSSLTCBS=8,  
SSLCACHE=CICS,  
SSLDELAY=600,
```

6.4.2 Defining a new TCPIP SERVICE

Since the inquireCatalog and inquireSingle service requests will not be subject to SSL processing, we need to create a new TCPIP SERVICE specifically for the placeOrder requests.

The following commands were issued to create a new TCPIP SERVICE for the server-side SSL scenario (Example 6-8).

Example 6-8 CEDA commands to create TCPIPSERVICE S3C1SSL

```
CEDA COPY GROUP(S3C1) TCPIPSERVICE(S3C1) AS(S3C1SSL)
CEDA ALTER GROUP(S3C1) TCPIPSERVICE(S3C1SSL) PORTNUMBER(14311)
CEDA ALTER GROUP(S3C1) TCPIPSERVICE(S3C1SSL) SSL(YES)
CEDA ALTER GROUP(S3C1) TCPIPSERVICE(S3C1SSL) CERTIFICATE(CIWSS3C1)
CEDA ALTER GROUP(S3C1) TCPIPSERVICE(S3C1SSL)
CIPHERS(05042F303132330A1613100D)
```

The results are shown in Example 6-9.

Example 6-9 S3C1SSL TCPIPSERVICE

```
CEDA View TCpipservice( S3C1SSL )
TCpipservice : S3C1SSL
GRoup       : S3C1
DEscription ==> CICS Example Application - TCPIPSERVICE
Urm        ==> NONE
Portnumber ==> 14311
STatus     ==> Open
PROtocol   ==> Http
TRansaction ==> CWXN
Backlog    ==> 00005
TSqprefix  ==>
Ippaddress ==>
S0cketclose ==> No
Maxdatalen ==> 000032
SECURITY
SS1       ==> Yes
CErtificate ==> CIWSS3C1
(Mixed Case)
PRIVacy    : Supported
CIphers   ==> 0A1613100D05042F30313233
AUthenticate ==> No
Realm      ==>
ATTachsec  ==>
DNS CONNECTION BALANCING
DNsgroup   ==>
GRPcritical ==> No
```

The new TCPIPSERVICE listens on **Portnumber** 14311. It is not possible to have the same port accepting both HTTP and HTTPS requests.

Protocol is set to `Http`, which means that this is a `TCPIPSERVICE` that expects messages in `HTTP` format. `TCPIP SERVICES` can also be configured to support Internet Inter-ORB Protocol (`IIOP`), External Call Interface (`ECI`), IP Connections (`IPIC`) and User-defined protocols.

Socketclose is set to `No`. `CICS` will never close a socket but will always wait for the client to close the socket. This enables persistent sockets, and reduces the overhead of opening and closing a socket for every request. We want to keep `SSL` handshaking to a minimum.

Maxdatalen defaults to 32 Kilobytes. This is the maximum amount of inbound data the `TCPIP SERVICE` can receive. This is a protection against too much data being sent to `CICS` as part of a denial of service attack. Our messages will not be this large. The maximum setting for `Maxdatalen` is 524,288 Kilobytes.

SSL is set to `Yes`. This means that this `TCPIP SERVICE` will use server-side `SSL`, sending its server certificate as part of the handshake.

Certificate is set to `CIWSS3C1`. This is the label of the `X.509` certificate that is used as a server certificate during the `SSL` handshake. If this attribute is omitted, the default certificate defined in the key ring for the `CICS` region user ID is used.

Ciphers is set to `0A1613100D05042F30313233`. We explain this setting in detail in the next section.

Authenticate is set to `No`. We do not require the client to authenticate with `CICS`.

Ciphers in `TCPIP SERVICES`

As we saw on page 198, the `SIT` parameter `ENCRYPTION` determines which Ciphers are available for `CICS` to use. However, unlike `WebSphere`, `ENCRYPTION=STRONG` means that all ciphers, whether strong, medium or weak are available to the `CICS` region.

As a default, when a `TCPIP SERVICE` uses `SSL`, the `CIPHERS` attribute is initialized with the full set of ciphers supported by the `CICS` region. The `CIPHERS` attribute can also be re-initialized to this value by deleting all ciphers within it and pressing `Enter`. The results can be seen in Example 6-10.

Example 6-10 Default Ciphers for `ENCRYPTION=STRONG`

```
SECURITY
SSL          ==> Yes          Yes | No | Clientauth
Certificate  ==> CIWSS3C1
(Mixed Case)
PRivacy     : Supported      Notsupported | Required | Supported
CIPHERS     ==> 050435363738392F303132330A1613100D0915120F0C03060201
```

The Ciphers are represented as two character codes, which relate to the list of ciphers referenced on page 198. Cipher suites 01 and 02 are highlighted because they are very weak and offer no encryption. It is unlikely that an SSL client would only support those algorithms, but to remain secure we will remove them from the list by re-specifying it, as shown below (Example 6-11).

Example 6-11 Removing ciphers without encryption from TCPIPSERVICE S3C1SSL

```
CEDA ALTER GROUP(S3C1) TCPIPSERVICE(S3C1SSL)
CIPHERS(050435363738392F303132330A1613100D0915120F0C0306)
```

Some ciphers have short keys, which are easier to crack. We therefore remove all ciphers from the list whose keys are less than 128 bits in length with the command shown in Example 6-12.

Example 6-12 Removing ciphers with keys less than 128 bits in length

```
CEDA ALTER GROUP(S3C1) TCPIPSERVICE(S3C1SSL)
CIPHERS(050435363738392F303132330A1613100D)
```

Other ciphers offer a great deal of cryptographic protection, but are expensive in terms of processor power, especially if the underlying hardware does not support them. On our system, the underlying hardware does not support AES with 256 bit keys, so we remove those ciphers from the list, as shown in Example 6-13.

Example 6-13 Removing ciphers with 256 bit keys and AES algorithm

```
CEDA ALTER GROUP(S3C1) TCPIPSERVICE(S3C1SSL)
CIPHERS(05042F303132330A1613100D)
```

Important: If there are no common cipher suites between the client and CICS, CICS will close the connection.

CICS also allows you the opportunity to alter the order of the cipher suites. CICS will work through the list from left to right to find the first cipher suite it has in common with the client.

Currently our TCPIPSERVICE will try all possible 128-bit key cipher suites first, then 168-bit keycipher suites. This is reasonable, as 128-bit key encryption is secure and less expensive than 168-bit key encryption. However, we decide that we'd prefer CICS to try the 168-bit suites first, and only use 128-bit suites if they are the only available common cipher suite. We issue the following command to reorder the cipher suites (Example 6-14).

Example 6-14 Reordering the cipher suites to prefer 168-bit keys

```
CEDA ALTER GROUP(S3C1) TCPIPSERVICE(S3C1SSL)
CIPHERS(0A1613100D05042F30313233)
```

Attention: These cipher suite choices are given as an example of the type of selection you can make. The cipher suites you choose to enable will depend on your specific security requirements and also the cryptographic hardware availability.

6.4.3 Defining a new WEBSERVICE

We want to secure the Web Service `placeOrder` by using SSL. We have set up two `TCPIPSERVICE`s in the CICS region, one using HTTP for the `inquireSingle` and the `inquireCatalog` Web services, and the other using HTTPS which we plan to use for the `placeOrder` Web Service.

However, if we utilize the automatic pipeline scanning mechanism to detect the `placeOrder.wsbind` file, it will also create a default `URIMAP`. The default `URIMAP` allows requests over HTTP to access the Web Service.

We move the `placeOrder.wsbind` file to a new directory, `/CIWS/S3C1/wsbind/provider/secured/`. This is not the same directory as `PIPELINE EXPIPE01` uses as its pickup directory, therefore CICS will not automatically create a `WEBSERVICE` resource.

The following commands are issued to define the `WEBSERVICE` resource (Example 6-15).

Example 6-15 Commands to create WEBSERVICE placeOrd

```
CEOT TRANIDONLY
CEDA DEFINE GROUP(S3C1) WEBSERVICE(placeOrd)
PIPELINE(EXPIPE01)
WSBIND(/CIWS/S3C1/wsbind/provider/secured/placeOrder.wsbind)
```

Tip: When entering mixed-case data without using the CEDA screens, be aware that the terminal's Upper-case Translation settings might alter what you type. A common default is for terminals to transform all characters to uppercase. Without `CEOT TRANIDONLY`, the `WSBIND` file would be `/CIWS/S3C1/WSBIND/PROVIDER/SECURED/PLACEORDER.WSBIND`. This would not be found, because UNIX Systems Services file names are case-sensitive.

The WEBSERVICE is called `placeOrd`. When CICS installs a Web Service as a result of a pipeline scan, the WEBSERVICE resource created has the same name as the `wsbind` file. For example, `inquireSingle.wsbind` would result in a WEBSERVICE resource of `inquireSingle`. When defining a WEBSERVICE resource in the CSD, the WEBSERVICE name is limited to a maximum of 8 characters. We therefore call the WEBSERVICE `placeOrd`.

The WEBSERVICE can run in the same PIPELINE as `inquireSingle` and `inquireCatalog` because the security is established at the transport level.

6.4.4 Defining a new URIMAP

The URIMAPs created by CICS following a pipeline scan take default values for the following parameters:

- ▶ **Scheme**(Http). This means that this URIMAP is valid for HTTP, HTTPS and MQ/JMS requests.
- ▶ **Tcpiptest**(`TCPIP`). This means that this URIMAP is valid for requests received by any TCPIP SERVICE.
- ▶ **Host**(*). This means that this URIMAP is valid for requests that specified any host, or that came from any MQ Queue.

We want to restrict access to the `placeOrder` Web Service to requests over HTTPS. We issue the following commands to create a URIMAP (Example 6-16).

Example 6-16 Commands to create a URIMAP which is HTTPS only

```
CEDA DEFINE GROUP(S3C1) URIMAP(placeOrd)
USAGE(PIPELINE)
SCHEME(HTTPS) HOST(*) PATH(/exampleApp/placeOrder)
user ID(PRIVAT01) TRANSACTION(ORDS)
PIPELINE(EXPIPE01) WEBSERVICE(placeOrd)
```

This URIMAP will match requests using HTTPS, for any host and any TCPIP SERVICE, with a path of `/exampleApp/placeOrder`. The pipeline task will run with a default user ID of `PRIVAT01`, under transaction `ORDS`, with pipeline `EXPIPE01` and `WEBSERVICE(placeOrd)`.

Note: We found that a Web service could be invoked using WebSphere MQ as transport even if the URIMAP specifies `SCHEME(HTTPS)`. If you want to restrict such access, you could specify a `TCPIP` SERVICE attribute for the URIMAP which restricts access to requests received via a specific TCPIP SERVICE.

6.5 Testing the SSL scenario

In order to test our server-side SSL configuration, we deploy the CICS supplied Web client of the catalog manager application (file **ExampleAppClientV6.ear**) to our WebSphere Application Server running on Windows. See “Installing and setting up the application” on page 174 for more information on deploying and testing the sample application.

We submit a placeOrder request so that a Web service request is sent from WebSphere Application Server to CICS.

In this section we show the results of testing the server-side SSL scenario:

- ▶ Request from WebSphere to CICS captured with TCP/IP Monitor
- ▶ CICS Trace showing cipher selection and handshake
- ▶ CICS error messages
- ▶ Browser response

Request from WebSphere to CICS

An example of the request message sent by WebSphere using SSL is shown in Figure 6-8.

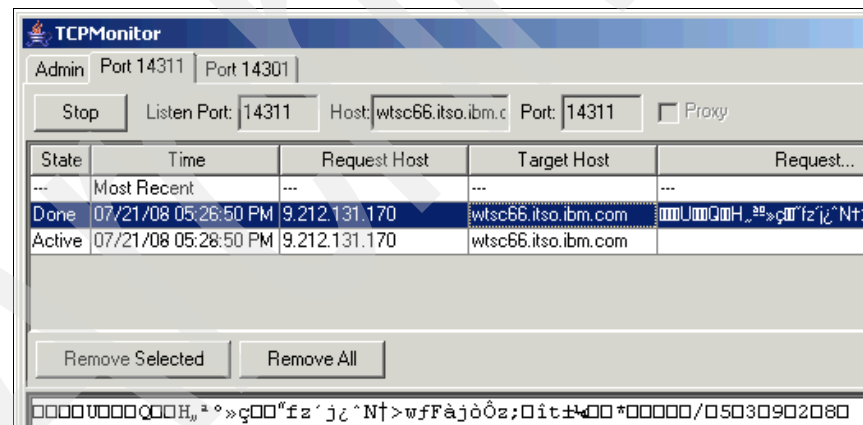


Figure 6-8 TCPMonitor shows that the data sent is unreadable

As can be seen in this figure, the data sent between WebSphere and CICS using SSL is unreadable. Our data is private and cannot be tampered with.

CICS trace showing cipher selection

Sockets Level 1 trace point SO 0802 below (Example 6-17) shows the results of the SSL handshake with WebSphere.

Example 6-17 Cipher suite 0A chosen in SSL handshake

```
SO 0802 SOSE EXIT - FUNCTION(SECURE_SOC_INIT) RESPONSE(OK)
GSK_RETURN_CODE(0) CERTIFICATE_user ID() CIPHER_SELECTED(0A)
CIPHER_NAME(SSL_RSA_WITH_3DES_EDE_CBC_SHA)
```

We can see that CICS has selected cipher 0A, which is the first cipher specified in the TCPIP SERVICE. The chosen cipher is SSL_RSA_WITH_3DES_EDE_CBC_SHA, which is one of the ciphers WebSphere supports in its Strong suite.

CICS error messages

The following messages are seen in the CICS region's MSGUSR log (Example 6-18).

Example 6-18 Error messages seen in MSGUSR during Server-side SSL scenario

```
DFHXS1111 07/21/2008 10:38:29 A6POS3C1 ORDS Security violation by user
PRIVAT01 for resource CIWS3D.ORDS in class TCICSTRN. SAF codes are
(X'00000008',X'00000000'). ESM codes are (X'00000008',X'00000000').
DFHWB0361 07/21/2008 10:38:29 A6POS3C1 An attempt to attach a CICS Web
alias transaction for user ID PRIVAT01 has failed because the user is
not authorized to execute transaction ORDS. Host IP address: 9.12.4.75.
Client IP address: 9.212.131.170. TCPIP SERVICE: S3C1SSL.
DFHAC2003 07/21/2008 10:38:29 A6POS3C1 Security violation has been
detected term id = ????, trans id = ORDS, user ID = PRIVAT01.
```

The Web Service request arrives in CICS, CICS finds the matching URIMAP, placeOrd, and starts a task with the following security credentials:

- ▶ Transaction id of ORDS
- ▶ User id of PRIVAT01

The user ID PRIVAT01 does not have access to transaction ORDS. This is intentional, because we only want to authress USERWS02 to the placeOrder Web Service.

Browser response

The error shown in Figure 6-9 can be seen in the browser.



Figure 6-9 Error seen in browser from SSL scenario

CICS cannot start the pipeline task with user ID PRIVAT01 and transaction ORDS, so the CICS Web Support component returns a HTTP 403 Forbidden message to WebSphere.

Although server-side SSL is now working correctly, we next need to configure SSL client authentication in order to authenticate and identify the Web service requester.

6.6 Creating the WebSphere client certificate

In this section, we document the following steps:

- ▶ Creating the WebSphere certificate
- ▶ Exporting the WebSphere certificate to a file
- ▶ Configuring WebSphere to send the client certificate as part of the SSL handshake

6.6.1 Creating the WebSphere certificate

In order to enable client-server SSL processing, we create a certificate and key pair to be used by WebSphere. The certificate and key pair are then stored in a key store.

In this section we create a self-signed certificate for WebSphere to use as a client certificate. We perform the following steps using the WebSphere admin console:

1. Select **Security** → **SSL certificate and key management**.
2. Under Related Items, select **NodeDefaultKeyStore**.

Under **Additional Properties**, click **Personal Certificates**. The results can be seen in Figure 6-10.

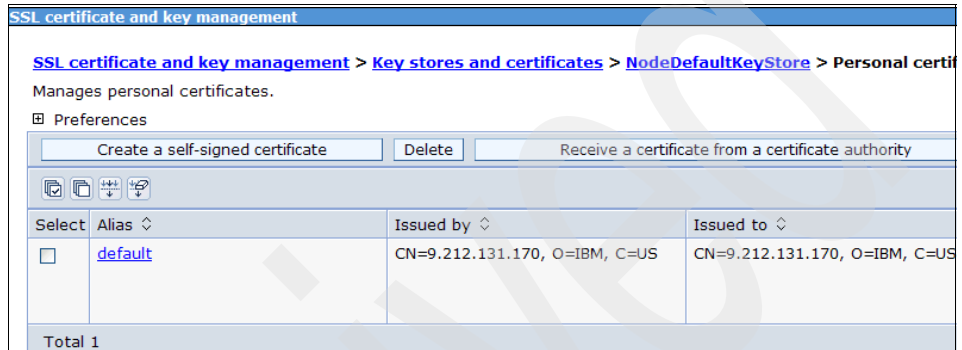


Figure 6-10 Personal certificates within the node default key store

3. Click **Create a self-signed certificate** and enter the certificate information (Figure 6-11) as follows:

Alias: **waswincert**
 Key size: **1024**
 Common name: **WASWINCert02**
 Validity Period: **365 (days)**
 Organization: **IBM**
 Organization unit: **ITSO**
 Locality: **Montpellier**
 State/Province: **Herault**
 Country: **FR**

[SSL certificate and key management](#) > [Key stores and certificates](#) > [NodeDefaultKeyStore](#) > [Personal certificates](#) > **New**

Manages personal certificates.

Configuration

General Properties

* Alias
waswincert

Version
X509 V3

Key size
1024 bits

* Common name
WASWINCert02

* Validity period
365 days

* Organization
IBM

Organization unit
ITSO

Locality
Montpellier

State/Province
Herauld

Zip code

Country or region
FR

Apply OK Reset Cancel

Figure 6-11 waswincert self-signed certificate creation screen

4. Click **OK** to create the certificate and save the configuration changes. The resulting certificate can be seen in Figure 6-12.



Figure 6-12 Self-signed certificate waswincert in the NodeDefaultKeyStore

6.6.2 Exporting the WebSphere certificate to a file

In this section we show how we extract the WebSphere certificate and its public key (the WebSphere private key remains secure in the key store).

1. In **SSL certificate and key management** → **Key stores and certificates** → **NodeDefaultKeyStore** → **Personal certificates**, as shown in Figure 6-12, select the `waswincert` certificate and then click **Extract**. The results can be seen in Figure 6-12.

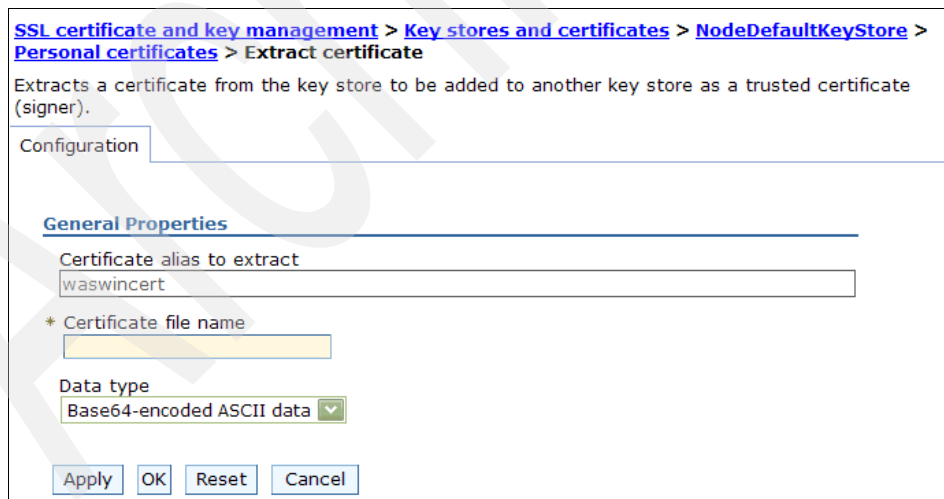


Figure 6-13 Exporting waswincert self-signed certificate

2. Specify the **Certificate file name** to export the certificate to, C:\waswincert.cer, and leave the **Data type** as Base64-encoded ASCII data. Click **OK**.

Checking the file location confirms that the file has been created and is in Base64 format.

6.6.3 Configuring WebSphere to send the client certificate in SSL

In this section we configure WebSphere to send the self-signed certificate as a client certificate during SSL handshaking.

We perform the following steps using the WebSphere admin console:

1. Select **Security** → **SSL certificate and key management**.
2. Under Related Items select **SSL configurations**.
3. Select **NodeDefaultSSLSettings**.
4. On the NodeDefaultSSLSettings screen, select the **Default client certificate alias** drop down box and select **waswincert**. The results are shown in Figure 6-14.

The screenshot shows the WebSphere admin console interface for the **NodeDefaultSSLSettings** configuration page. The breadcrumb navigation at the top reads: **SSL certificate and key management > SSL configurations > NodeDefaultSSLSettings**. Below the breadcrumb, there is a description: "Defines a list of Secure Sockets Layer (SSL) configurations." and a "Configuration" tab. The page is divided into two main sections: **General Properties** and **Additional Properties**.

General Properties:

- Name:** NodeDefaultSSLSettings
- Trust store name:** NodeDefaultTrustStore
- Keystore name:** NodeDefaultKeyStore (with a "Get certificate aliases" button next to it)
- Default server certificate alias:** default
- Default client certificate alias:** waswincert
- Management scope:** (cell):Cev8-Pc12Node01Cell:(node):Cev8-Pc12Node01

Additional Properties:

- [Quality of protection \(QoP\) settings](#)
- [Custom properties](#)

At the bottom of the form, there are four buttons: **Apply**, **OK**, **Reset**, and **Cancel**.

Figure 6-14 waswincert certificate selected as Default client certificate alias

5. Click **Apply**, then click **Save** to save the changes to the master configuration.

6.7 Configuring the CICS service provider for SSL client authentication

In this section we configure CICS to request a client certificate during SSL handshaking, and to use that client certificate to authenticate the requester.

We document the following steps:

- ▶ Configuring the TCPIP SERVICE for SSL client authentication
- ▶ Adding the client certificate to RACF
- ▶ Authorizing the service requester to run the CICS Web Service

6.7.1 Configuring the TCPIP SERVICE for SSL client authentication

There are two parameters that have to be changed on the TCPIP SERVICE S3C1SSL in order to activate client authentication. The following command is issued (Example 6-19).

Example 6-19 Commands to configure TCPIP SERVICE for SSL client authentication

```
CEDA ALTER GROUP(S3C1) TCPIP SERVICE(S3C1SSL)  
SSL(CLIENTAUTH) AUTHENTICATE(CERTIFICATE)
```

The SSL attribute of the TCPIP SERVICE is now set to CLIENTAUTH. CICS now requires connections to this TCPIP SERVICE to provide a client certificate.

The AUTHENTICATE attribute is now set to CERTIFICATE. CICS uses the client certificate to determine the user ID under which CICS transactions will run.

Note: Authenticate(CERTIFICATE) can only be used with SSL(CLIENTAUTH), however, specifying CLIENTAUTH does not mandate the use of Authenticate(CERTIFICATE). You can have valid reasons for requiring a client certificate but not wanting to use it for authentication.

The TCPIP SERVICE is closed and then reinstalled with the following commands (Example 6-20).

Example 6-20 Commands to reinstall the amended TCIPSERVICE

```
CEMT SET TCIPSERVICE(S3C1SSL) CLOSED
CEDA INSTALL GROUP(S3C1) TCIPSERVICE(S3C1SSL)
```

The results are shown in Example 6-21.

Example 6-21 Amended TCIPSERVICE as seen by CEMT INQUIRE TCIPSERVICE

```
INQUIRE TCIPSERVIC(S3C1SSL)
STATUS: RESULTS - OVERTYPE TO MODIFY
  TcpiPs(S3C1SSL ) Ope Por(14311) Http Cl i Tra(CWXN) Cer
    Con(00000) Bac( 00005 ) Max( 000032 ) Urm( NONE      ) Sup
```

As can be seen, this TCIPSERVICE has the attribute **Cl i**, which indicates that it is using SSL client authentication. In addition, it has the attribute **Cer**, which indicates that it is using certificate-based authentication. The detailed view is shown in Example 6-22.

Example 6-22 Detailed view of TCIPSERVICE S3C1SSL.

```
INQUIRE TCIPSERVIC(S3C1SSL)
RESULT - OVERTYPE TO MODIFY
  TcpiPservic(S3C1SSL)
  Openstatus( Open )
  Port(14311)
  Protocol(Http)
  Ssltype(Clientauth)
  Transid(CWXN)
  Authenticate(Certificat)
  Connections(00000)
  Backlog( 00005 )
  MaxdataLen( 000032 )
  Urm( NONE      )
  Privacy(Supported)
  Ciphers(0A1613100D05042F30313233)
  Ippaddress(9.12.4.75)
  Socketclose(Wait)
  ClOsetimeout(000000)
  Realm()
  DnsgrOup()
```

Here we can see clearly that:

- ▶ We are using SSL client authentication.
- ▶ We use the client certificate for authentication.

6.7.2 Adding the client certificate to RACF

In this section we add the client certificate RACF and specify the RACF ID which is to be associated with the certificate. We take the following steps:

1. FTP the `waswincert.cer` file to the z/OS system.
2. Import the certificate into RACF and map to a valid user ID.

FTP the `waswincert.cer` file to z/OS

The following commands are issued to transfer the `waswincert.cer` file to the host (Example 6-23).

Example 6-23 FTP WebSphere certificate to z/OS

```
C:\>ftp wtsc66.itso.ibm.com
Connected to wtsc66.itso.ibm.com.
User (wtsc66.itso.ibm.com:(none)): CICSRS6
331-Password:
230-220-FTPMVS1 IBM FTP CS V1R9 at wtsc66.itso.ibm.com, 08:17:08 on 2008-07-08.
230-CICSRS8 is logged on. Working directory is "CICSRS8.".
ftp> cd ..
250 "" is the working directory name prefix.
ftp> cd CIWS
250 "CIWS." is the working directory name prefix.
ftp> asc
200 Representation type is Ascii NonPrint
ftp> quote site lr=84 rec=vb
200 SITE command was accepted
ftp> put waswincert.cer waswin02.cer
200 Port request OK.
125 Storing data set CIWS.WASWIN02.CER
250 Transfer completed successfully.
ftp: 926bytes received in 0.00Seconds 2712000.00Kbytes/sec.
ftp> bye
221 Quit command received. Goodbye.
```

- ▶ Base 64 encoded data is in a text format, not a binary format, so we must specify `asc`. When the data is sent to the mainframe, it will be converted to EBCDIC, which is what is required.
- ▶ RACF expects datasets for certificates to be Variable Blocked and 84 bytes in length. The `quote site` command allows site specific commands to be sent. Here we inform the FTP server that files created should be of this format.
- ▶ The `put` command sends the first file and asks the FTP server to create a new file with the second name. `waswincert.cer` does not comply with MVS™ dataset naming conventions, therefore we rename it.

- ▶ The prefix is added to the front of the file name. The complete data set name is CIWS.WASWIN02.CER. This file is shown in Example 6-24.

Example 6-24 CIWS.WASWIN02.CER on z/OS

```

BROWSE      CIWS.WASWIN02.CER                      Line 00000000 Col 001 080
  Command ==>                                     Scroll ==> PAGE
***** Top of Data *****
-----BEGIN CERTIFICATE-----
MIICdjCCAd+gAwIBAgIESI9wJANBgkqhkiG9w0BAQUFADBOMQswCQYDVQQGEwJGUJjEJMAcGA1UE
ERMAMRAwDgYDVQQIEwZIZXJhdWx0MRQwEgYDVQQHEwtNb250cGVsbG11cjEMMAoGA1UEChMDSUJN
MQ0wCwYDVQQLEwRJVFNPMRUwEwYDVQQDEwXQVNXSU5DZXJOMDIwHhcNMDgwNzIOMDgzMDU4WhcN
MDkwNzIOMDgzMDU4WjBOMQswCQYDVQQGEwJGUJjEJMAcGA1UEERMAwDgYDVQQIEwZIZXJhdWx0
MRQwEgYDVQQHEwtNb250cGVsbG11cjEMMAoGA1UEChMDSUJNMQ0wCwYDVQQLEwRJVFNPMRUwEwYD
VQQDEwXQVNXSU5DZXJOMDIwZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBALdf+1qh51cktQTX
we7tcJ+wTeDnsbEPEppNvTqm4Y8I1hj6BtTZBbmXxi5Sbw8/Z1QcgPMszN6TcL7pVZzTfqXwYh7
yVeejWsnK/kA11hdWY9W2L718cj0zcv2MWWNj0YNGV0CPyNDUHHWxv8/ELX0dj11GgWwvTbfU56
7KdnAgMBAAGjFTATMBEGA1UdDgQKBAhNVLc6Kn1DojANBgkqhkiG9w0BAQUFAA0BgQCW4nUFRu6
+LVxDx+KotDQ2h6VgUfuRoNaIdM10ive6Pf9JB4sAxSivo2mbSaPSsrdMKxR9xtsmNjHxYnRfCUR
JC2bQpDf3t5SR+IHiaHY+kfQRP+Z/1UmZspQ+OZH1zzt0kYN4xu6nr/rQ611AkacfiIsDFJENBkn
7v1sTkqLIA==
-----END CERTIFICATE-----
***** Bottom of Data *****

```

Import the certificate into RACF and map to a valid user ID

The certificate now needs to be imported into RACF and associated with a RACF user ID. The following commands are issued (Example 6-25).

Example 6-25 Adding the waswincert certificate to RACF command

```

RACDCERT ID(USERWS02) ADD(CIWS.WASWIN02.CER)
WITHLABEL('waswincert')
TRUST

```

This command:

- ▶ Adds the certificate to RACF.
- ▶ Associates the certificate with RACF user ID USERWS02.
- ▶ Creates a certificate label of waswincert.
- ▶ Indicates that the certificate can be used to authenticate a user ID with the TRUST option.

6.7.3 Authorizing the service requester

In this section we show the RACF commands that are used to authorize the service requester to run the placeOrder Web service.

Permitting access

Example 6-26 shows the RACF commands that we used to permit the service requester's user ID (USERWS02) to start the ORDS transaction.

Note: This scenario does not use the ORDR transaction. This is because the security context for the Web Service is established using transport-level security, not message-level security.

Example 6-26 RACF command to allow USERWS02 to run ORDS transaction

```
PERMIT CIWS3D.ORDS CLASS(TCICSTRN) ID(USERWS02) ACCESS(READ)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

6.8 Testing the SSL client authentication scenario

In this section we show the results of testing the scenario. We submit a placeOrder request so that a Web service request is sent from WebSphere Application Server to CICS.

We show the following:

- ▶ CEMT showing that the task runs under the correct user ID
- ▶ CICS Trace showing Client Certificate selection

CEMT showing SSL client authentication

As can be seen in Example 6-27 below, Task 67 is running under transaction ID ORDS and user ID USERWS02.

Example 6-27 ORDS running with user associated with client certificate

```
I TAS
STATUS: RESULTS - OVERTYPE TO MODIFY
Tas(0000067) Tra(ORDS)          Sus Tas Pri( 001 )
  Sta(U ) Use(USERWS02) Uow(C2BE61BC08BB7B44) Hty(EDF  )
Tas(0000069) Tra(CEDF) Fac(E004) Sus Ter Pri( 001 )
  Sta(SD) Use(CICSR8 ) Uow(C2BE61BC0952EB8C) Hty(ZCIOWAIT)
```

RESPONSE: NORMAL

SYSID=S3C1 APPLID=A6POS3C1
TIME: 05.30.41 DATE: 07.25.08

Important: User PRIVAT01 is specified as the user ID attribute on the URIMAP definition, however, it is not used for running ORDS because a user ID obtained from transport-level security overrides any user ID specified on the URIMAP.

CICS trace showing client certificate selection

Example 6-28 shows the CICS Sockets Level 1 Trace Point SO 0802 on exit from the SECURE_SOC_INIT function. We can see that a client certificate has been sent and associated with a RACF id because the CERTIFICATE_user ID field contains USERWS02.

We can also see that the cipher selected was 0A and that the cipher name was SSL_RSA_WITH_3DES_EDE_CBC_SHA. This is the same cipher suite as was selected in the Server-side SSL scenario.

Example 6-28 CICS Sockets Level 1 Trace showing Cipher and Certificate selection

```
SO 0802 SOSE EXIT - FUNCTION(SECURE_SOC_INIT) RESPONSE(OK)
GSK_RETURN_CODE(0) CERTIFICATE_user ID(USERWS02)
CIPHER_SELECTED(0A) CIPHER_NAME(SSL_RSA_WITH_3DES_EDE_CBC_SHA)
```

Archived

Signing the SOAP message

To provide integrity on the request and response messages, we can add an XML digital signature to the message. CICS supports XML digital signatures in both service provider and service requester modes.

In this chapter we show how a SOAP message can be signed by WebSphere Application Server and how the signature can be verified by CICS. We also show how the response message can be signed by CICS and verified by WebSphere.

We provide step-by-step security configurations for configuring the CICS service provider pipeline and the service requester J2EE application using the WebSphere Application Server Toolkit.

7.1 Scenario overview

In this section we show how a SOAP request message can be signed by WebSphere Application Server and how the signature can be verified by CICS, and how the response message can be signed by CICS and verified by WebSphere.

- ▶ The SOAP request message contains an X.509 certificate (the WebSphere certificate) that is used to sign the body of the message.
- ▶ The SOAP response message also contains an X.509 certificate (the CICS certificate) that is used to sign the body of the message.

The X.509 certificates are transported within a *BinarySecurityToken* element.

Important: There is a significant performance overhead when using WS-Security and XML digital signatures. See 2.6, “Comparison of transport level and SOAP message security” on page 61 for recommendations on choosing between WS-Security and SSL when implementing an integrity security solution.

We also show how the WebSphere certificate can be used to authenticate the service requester. The CICS-supplied security handler verifies the signature and uses the certificate to determine the user ID under which the CICS pipeline transaction will be run. This scenario is shown in Figure 7-1.

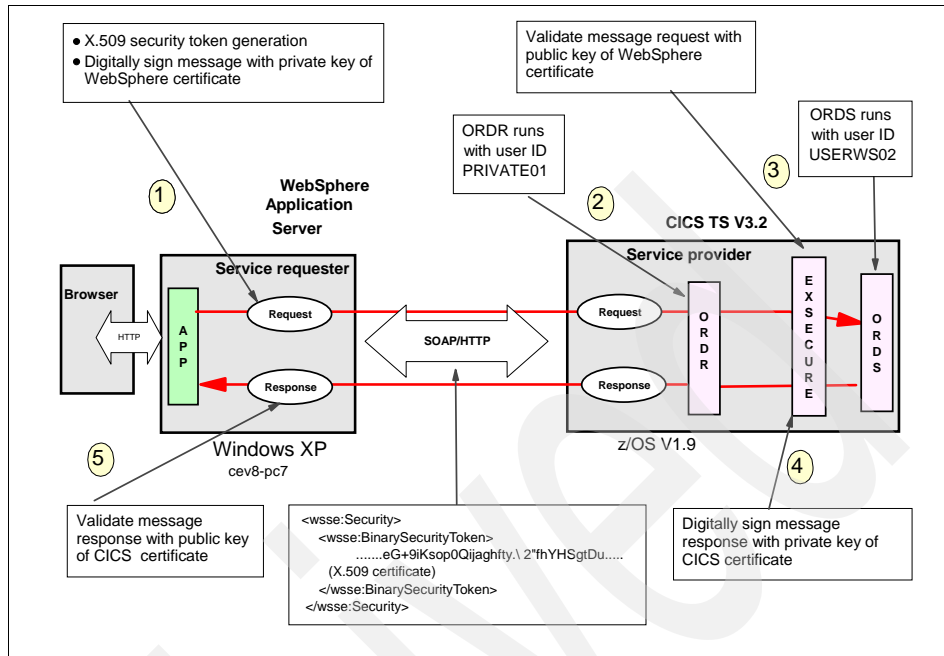


Figure 7-1 XML digital signature scenario

The sequence of steps is as follows:

1. The service requester (WebSphere in our example) generates a BinarySecurityToken element from the WebSphere X.509 certificate. It then signs the message with its private key so the service provider (CICS in our example) knows that the message can only be sent by the WebSphere application server.
2. The CICS region receives the SOAP Request and looks up the matching URIMAP. The URIMAP instructs CICS that requests for the path /exampleApp/placeOrder are to use the WEBSERVICE placeOrder, the PIPELINE EXSECURE, the transaction ORDR and the user ID PRIVATE01. CICS starts pipeline processing using this information.
3. The EXSECURE pipeline includes the DFHWSSE1 message handler which validates the signature in the SOAP message with the WebSphere public key. CICS calls RACF to map the WebSphere X.509 certificate to a RACF user ID and puts the mapped user ID USERWS01 in the DFHWS-USERID container. The pipeline also includes the user-written message handler CIWSMSGO which sets the context container DFHWS-TRANID to contain ORDS for any placeOrder request.

The ORDS transaction therefore runs under the caller's user ID USERWS01, and all further resource security checks for this request (for example, program and database security) are then performed against the USERWS01 user ID.

4. After the target program finishes processing, CICS signs the response message with its private key so that WebSphere knows that the message can only be sent by the CICS region.

Note: In this scenario we sign both the request and response messages. It is also possible to sign only the request, or to sign only the response.

5. WebSphere validates the signature with the public key of CICS.

7.2 Preparation

In this section we provide a summary of the software used in the scenario and the main definitions used for the CICS and WebSphere servers.

7.2.1 Software checklist

The software we used is listed in Table 7-1.

Table 7-1 Software used in the XML digital signature scenario

Windows	z/OS
Internet Explorer V7.0	z/OS V1.9
Windows XP Professional Version 2002 SP3	CICS Transaction Server V3.2
IBM WebSphere Application Server V6.1.0.17	
IBM WebSphere Application Server Toolkit V6.1.1.6	
Our J2EE applications <ul style="list-style-type: none"> ▶ ExampleAppClientV6.ear The Web client for the CICS catalog example application supplied with CICS TS. 	Our user-written CICS message handler programs: <ul style="list-style-type: none"> ▶ CIWSMSGO This program changes the transaction ID for placeOrder requests to ORDS. ▶ SNIFFER (message handler program). This program browses through the containers available in the pipeline.

7.2.2 Definition checklist

The definitions we used are listed in Table 7-2.

Table 7-2 Settings used in the XML digital signature scenario

Value	CICS TS	WebSphere Application Server
IP name	wtsc66.itso.ibm.com	cev8-pc7
TCP/IP port	14312	9080
Jobname	CIWSS3C2	
APPLID	A6POS3C2	
TCPIPSERVICE	S3C2	
Provider PIPELINES	EXPIPE01 for inquireSingle and inquireCatalog services EXSECURE for placeOrder service	
Configuration files	ITSO_7658_basicsoap12provider.xml for inquireSingle and inquireCatalog services ITSO_7658_wssec_signature_provider.xml for placeOrder service	

The certificates we used in our configuration are listed in Table 7-3.

Table 7-3 Certificates used in the XML digital signature scenario

Value	Format	File
CICS server certificate	PKCS12DER	CIWSS3C2.P12
CICS CA certificate	CERTDER	CIWSROOT.CER
WebSphere CA certificate	CERTDER	WASWINROOT.CER
WebSphere server certificate	PKCS12DER	WWSERV01.P12

The user IDs we used in our configuration are listed in Table 7-4.

Table 7-4 User IDs used in the XML digital signature scenario

Value	CICS TS
CICS region user ID	CIWS3D
User ID for which we want to permit access to inquireSingle and inquireCatalog services	PUBLIC01
User ID for which we want to permit access to the ORDR transaction of the placeOrder service	PRIVAT01
User ID for which we want to permit access to the ORDS transaction of the placeOrder service	USERWS01

Because the Catalog application inquireSingle and inquireCatalog services are read-only services, whereas the placeOrder service updates the database, we show here how to secure the placeOrder service.

We run the inquireSingle and inquireCatalog services under a predefined user ID PUBLIC01 that is defined in inquireSingle and inquireCatalog wsbind files.

We document the following procedures to enable support for XML signature processing:

- ▶ Setting up certificates and key pairs
- ▶ Configuring the WebSphere service requester for XML digital signature processing
- ▶ Configuring CICS for XML digital signature processing
- ▶ Testing the XML digital signature test scenario

7.3 Setting up certificates and key pairs

Figure 7-2 shows the certificates and key pairs used in the XML digital signature scenario.

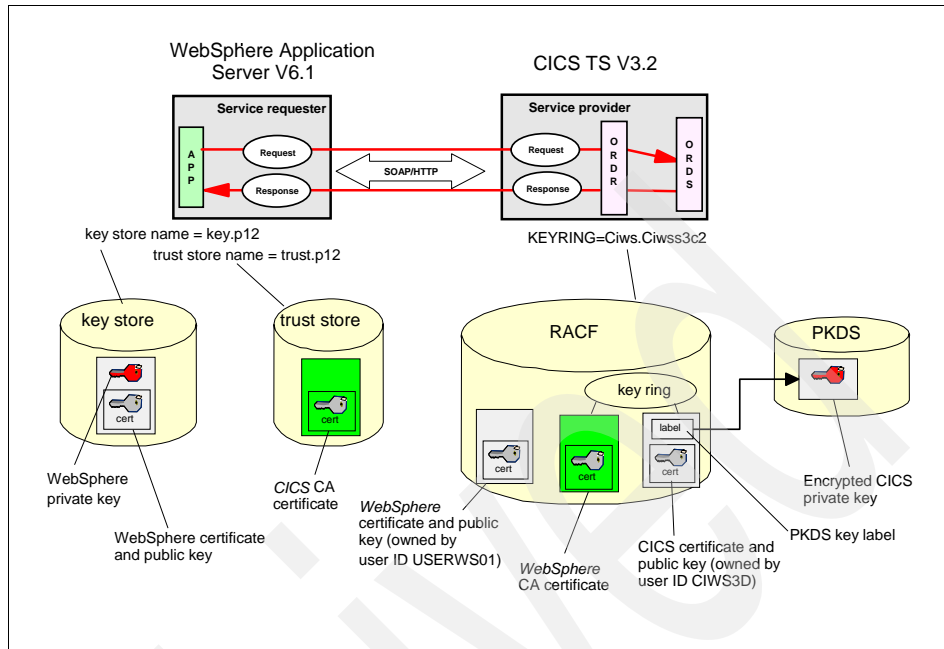


Figure 7-2 Certificates and key pairs used in XML digital signature scenario

Figure 7-2 shows the following stores for our certificates and key pairs:

► RACF:

- RACF contains the CICS certificate and holds the CICS public key. The associated private key is stored in a Private Key Data Set (PKDS).

In a public key cryptographic system, it is a priority to maintain the security of the private key. It is vital that only the intended user or application have access to the private key.

ICSF and the cryptographic hardware features ensure this by enciphering PKA private keys under a unique PKA object protection key. The PKA object protection key has itself been enciphered under a PKA *master key*.

Each PKA private key also has a name that is cryptographically bound to the private key and cannot be altered. ICSF uses the private key name or the *PKDS key label* to control access to the private key. This combination of hardware-enforced coupling of cryptographic protection and access control, through the use of RACF, is unique to ICSF. It provides a significant level of security and integrity for PKA applications.

The CICS certificate is connected to the key ring Ciws.Ciws3c2.

- RACF contains the WebSphere certificate which is used to identify the service requester.

Note: In this scenario we use the WebSphere certificate to authenticate the service requester. It must be present in the RACF database so that it can be mapped to a RACF user ID.

- RACF also contains the CICS and WebSphere CA certificates.
- ▶ The WebSphere key store:
 - The WebSphere key store contains the WebSphere certificate and holds the WebSphere public and private key pair. The WebSphere certificate is used to sign the SOAP request message.
- ▶ The WebSphere trust store:
 - The WebSphere trust store contains the CICS CA certificate that is used to validate the CICS certificate which is used to sign the SOAP response message.

In this section, we document the following steps:

1. Validating the cryptography hardware environment
2. Creating the CICS certificate and connecting the certificate to the keyring used by the CICS region
3. Creating the WebSphere certificate
4. Importing the WebSphere certificate to the key store
5. Adding the CICS CA certificate to the WebSphere trust store

7.3.1 Validating the cryptography hardware environment

CICS support for WS-Security signature and encryption processing benefits from the hardware acceleration capabilities of System z. It is a requirement, therefore, to run CICS on a system that has the appropriate cryptographic hardware configured. The specific cryptographic hardware requirement is dependent on the server type and z/OS level. See 4.2, “ICSF” on page 138 for more information about cryptographic hardware requirements.

The SC66 host system used for our XML signature test scenarios was a z9 EC (Enterprise Class) with 18 CPs. The SC66 had the following cryptographic hardware functions enabled:

- ▶ CP Assist for Cryptographic Function (CPACF)
- ▶ Crypto Express2 (CEX2) feature

Three CEX2C coprocessors and one CEX2A accelerator were configured.

Important: The CEX2 feature requires ICSF to be active.

Integrated Cryptographic Service facility

The Integrated Cryptographic Service Facility (ICSF) is a software element of z/OS that works with hardware cryptographic features and RACF to provide secure, high-speed cryptographic services in the z/OS environment. See 4.2, “ICSF” on page 138 for more information.

Example 7-1 shows the startup messages of ICSF on the SC66 system. It shows that 3 CEX2C coprocessors and 1 CEX2A coprocessor are configured.

Example 7-1 ICSF startup messages

```
S CSF
IEF695I START CSF WITH JOBNAME CSF IS ASSIGNED TO USER STC , GROUP SYS1
IEF403I CSF - STARTED - ASID=009A - SC66
CSFM441I CRYPTO EXPRESS2 COPROCESSOR E01, SERIAL NUMBER 94000264, ACTIVE
CSFM441I CRYPTO EXPRESS2 COPROCESSOR E02, SERIAL NUMBER 95001434, ACTIVE
CSFM441I CRYPTO EXPRESS2 COPROCESSOR E03, SERIAL NUMBER 95001437, ACTIVE
CSFM435I CRYPTO EXPRESS2 ACCELERATOR F00 IS ACTIVE
CSFM001I ICSF INITIALIZATION COMPLETE
CSFM400I CRYPTOGRAPHY - SERVICES ARE NOW AVAILABLE.
```

ICSF must be configured and started in order to create public/private key pairs to be used by CICS for XML signature or encryption processing.

7.3.2 Creating the CICS certificate

In order to enable XML signature processing, we create a certificate and key pair to be used by CICS, and then connect the certificate to the CICS key ring.

Example 7-2 shows the RACF command that was used to create the CICS certificate with the ICSF option.

Example 7-2 CICS certificate (CIWSS3C2)

```
RADCERT ID (CIWS3D) GENCERT
SUBJECTSDN(CN('CIWSS3C2'))
T('CIWSS3C2-certificate')
OU('ITS0')
O('IBM')
L('Poughkeepsie')
SP('New York')
C('US'))
```

```
WITHLABEL('CIWSS3C2')
SIGNWITH(CERTAUTH LABEL('CIWSROOT'))
KEYUSAGE(DOCSIGN)
NOTAFTER(DATE(2010/12/31))
SIZE(1024)
ICSF
PASSWORD('PASSWORD')
```

In Example 7-2:

1. The RACDCERT GENCERT command creates the certificate and a public/private key pair.
2. ID specifies that the CICS region user ID CIWS3D is to be associated with the certificate.
3. SUBJECTDSN specifies the subject's X.509 distinguished name.
4. WITHLABEL specifies the label name to be associated with the certificate.
5. SIGNWITH specifies the certificate with a private key that is signing the certificate. The CICS certificate is signed with the CICS CA certificate that was created in "Creating the CICS CA certificate" on page 163.
6. KEYUSAGE specifies how the keys associated with the certificate are to be used. We specify DOCSIGN because the certificate will be used for signing.
7. NOTAFTER specifies the date after which the certificate is no longer valid.
8. SIZE specifies the size of the private key expressed in decimal bits. We specified a high strength key length of 1024.
9. ICSF specifies that the private key to be generated is an RSA key to be stored in the ICSF PKDS (public key data set).

A certificate that CICS uses to sign SOAP messages (or decrypt encrypted SOAP messages) must be created with the ICSF option and ICSF must be configured and started in order for the certificate to be created or used by CICS. If ICSF is not operational, or does not have access to any cryptographic coprocessors or accelerators, you will receive the message ICSF is not operational when attempting to create the X.509 certificate.

Important: A certificate that CICS uses to sign SOAP messages, or to decrypt encrypted SOAP messages, must be created with the ICSF option.

Example 7-3 shows the RACF commands that we used to connect the CICS X.509 certificate to the key ring and to list the certificate.

Example 7-3 RACF command to connect CICS certificate to RACF key ring

```
RACDCERT ID(CIWS3D) CONNECT(ID(CIWS3D) LABEL('CIWSS3C2') RING(Ciws.Ciwss3c2))
RACDCERT ID(CIWS3D) LIST
```

Example 7-4 shows the output of the RACDCERT LIST command.

Example 7-4 Listing of CICS certificate (CIWSS3C2)

Digital certificate information for user CIWS3D:

Label:CIWSS3C2
Certificate ID:2QbDyebi88TDyebi4vPD8kBA
Status:TRUST
Start Date:2008/07/10 00:00:00
End Date: 2010/12/31 23:59:59
Serial Number:0B

Issuer's Name:CN=CIWSROOT.T=CIWSROOT-certificate.OU=ITSO.O=IBM.L=Poughkeepsie.SP=New York.C=US

Subject's Name:CN=CIWSS3C2.T=CIWSS3C2-certificate.OU=ITSO.O=IBM.L=Poughkeepsie.SP=New York.C=US

Key Usage:DOCSIGN
Private Key Type:ICSF
Private Key Size:1024
PKDS Label:IRR.DIGTCERT.CIWS3D.SC66.C2AB97874BD15B8B

Ring Associations: Ring Owner: CIWS3D
Ring:
>Ciws.Ciwss3c2<

Note: The RACF ISPF interface is an alternative to using RACF commands to create X.509 certificates.

Specifying the keyring as a SIT parameter

In addition to the basic security related SIT parameters that we configured in “Setting up basic security configuration” on page 161, we now specify the KEYRING=Ciws.Ciwss3c2 for our CICS region. This specifies the name of a key ring we created in the RACF database that contains keys and certificates used by this CICS region. The CICS region is restarted in order to activate the new SIT parameter.

7.3.3 Creating the WebSphere certificate

In order to enable XML signature processing, we create a certificate and key pair to be used by WebSphere. The certificate and key pair are then stored in a key store.

First we use the RACDCERT command to create the certificate in RACF. The command is listed in Example 7-5.

Example 7-5 RACDCERT command to create WebSphere certificate

```
RACDCERT ID(USERWS01) GENCERT          +
SUBJECTSDN(CN('WASWINCert01'))        +
T('CWASWINCert01-certificate')        +
OU('ITSO')                              +
O('IBM')                                +
L('Montpellier')                       +
SP('Herault')                           +
C('France'))                            +
WITHLABEL('WASWINServ01')              +
SIGNWITH(CERTAUTH LABEL('WASWINROOT')) +
NOTAFTER(DATE(2010/12/31))              +
SIZE(1024)
```

Next we export the certificate to a dataset, and use the OPUT command to copy the exported certificate to a HFS file, from where it can be imported by the client. The commands we use are listed in Example 7-6

Example 7-6 Export and copy the certificates to a HFS file

```
RACDCERT ID(USERWS01) EXPORT(LABEL('WASWINServ01')) +
DSN('CIWS.WWSERV01.P12') FORMAT(PKCS12DER)
OPUT 'CIWS.WWSERV01.P12' '/CIWS/Certificates/WWSERV01.P12' +
BINARY CONVERT(NO)
```

7.3.4 Importing the WebSphere certificate to the key store

Next we add the WebSphere certificate and key pair to the WebSphere key store. We used the default WebSphere key store NodeDefaultKeyStore. The default references file key.p12 in the node or cell directories of the configuration repository. This file is a PKCS12 type key store.

Table 7-5 shows the main settings for the key store and the WebSphere certificate.

Table 7-5 Client key store and certificate values

Field	Value
key store name	NodeDefaultKeyStore
Filename	key.p12
Storepass	WebAS
Storetype	PKCS12
Distinguished Name	CN=WASWINCert01, OU=ITSO, O=IBM
Key size	1024
Alias	waswinserv01

In order to configure WebSphere to use this certificate, we copied the WebSphere certificate from the host and added the certificate to the key store using the WebSphere Administration Console.

Example 7-7 shows how we used FTP to send the exported certificate (in binary format) to our WebSphere machine.

Example 7-7 FTP WebSphere certificate to WebSphere machine

```
C:\>ftp wtsc66.itso.ibm.com
Connected to wtsc66.itso.ibm.com.
User (wtsc66.itso.ibm.com:(none)): CICSRS6
331-Password:
230-220-FTPMVS1 IBM FTP CS V1R9 at wtsc66.itso.ibm.com, 08:17:08 on 2008-07-08.
230-CICSRS6 is logged on. Working directory is "CICSRS6.".
ftp> cd /CIWS/certificates
250 HFS directory /CIWS/certificates is the current working directory.
ftp> bin
200 Representation type is Image
ftp> get WWSERV01.P12
200 Port request OK.
125 Sending data set /CIWS/certificates/WWSERV01.P12
250 Transfer completed successfully.
ftp: 2712 bytes received in 0.00Seconds 2712000.00Kbytes/sec.
ftp> bye
221 Quit command received. Goodbye.
```

We then perform the following steps using the WebSphere admin console:

1. Select **Security** → **SSL certificate and key management**.
2. Under Related Items, select **Key stores and certificates**.
3. The resulting panel shows the default key stores and trust stores. Click the **NodeDefaultKeyStore**.
4. Under Additional Properties, select **Personal certificates**, click **Import** and enter these values:

Key file name: c:\WWSERV01.P12
Type: PKCS12
Key file password: PASSWORD

5. Click **Get key file aliases**. Figure 7-3 shows the certificate alias that WebSphere has found in the key file.



Figure 7-3 Import WebSphere certificate into key store

6. Click **OK** and the WebSphere certificate is imported into the key store.
7. Save the changes.

Figure 7-4 shows the WebSphere certificate imported into the NodeDefaultKeyStore.

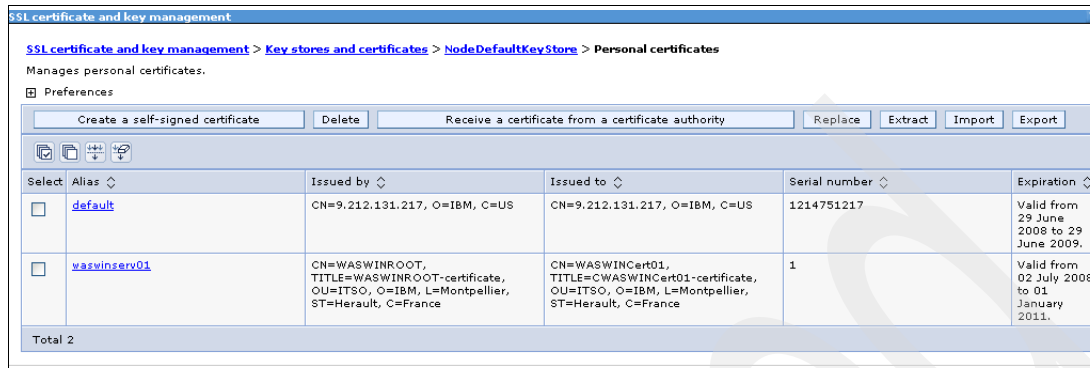


Figure 7-4 Key store certificates

7.3.5 Adding the CICS CA certificate to the trust store

To complete the setup of certificates, we now add the CICS CA certificate to the WebSphere trust store. WebSphere uses this CA certificate to validate the CICS certificate which is used to sign the message response.

We used the default WebSphere trust store NodeDefaultTrustStore. The default references file trust.p12 in the node or cell directories of the configuration repository. This file is a PKCS12 type trust store.

Table 7-6 shows the main settings for the trust store and the WebSphere certificate.

Table 7-6 Client trust store and certificate values

Field	Value
trust store name	NodeDefaultTrustStore
Filename	trust.p12
Storepass	WebAS
Storetype	PKCS12
Distinguished Name	CN=CIWSR00T, OU=ITSO, O=IBM
Key size	1024
Alias	cics ca certificate

We FTPed the CICS CA certificate from the host (in binary mode) and added the certificate to the trust store using the WebSphere Administration Console.

We perform the following steps using the WebSphere admin console:

1. Select **Security** → **SSL certificate and key management**.
2. Under Related Items, select **Key stores and certificates**.
3. The resulting panel shows the default key stores and trust stores. Click the **NodeDefaultTrustStore**.
4. Under Additional Properties, select **Signer certificates**, click **Add** and enter these values:

Alias: CICS CA certificate
File name: c:/CIWSROOT.CER
Data type: Binary DER file

Figure 7-5 shows the CICS CA certificate being added to the trust store.

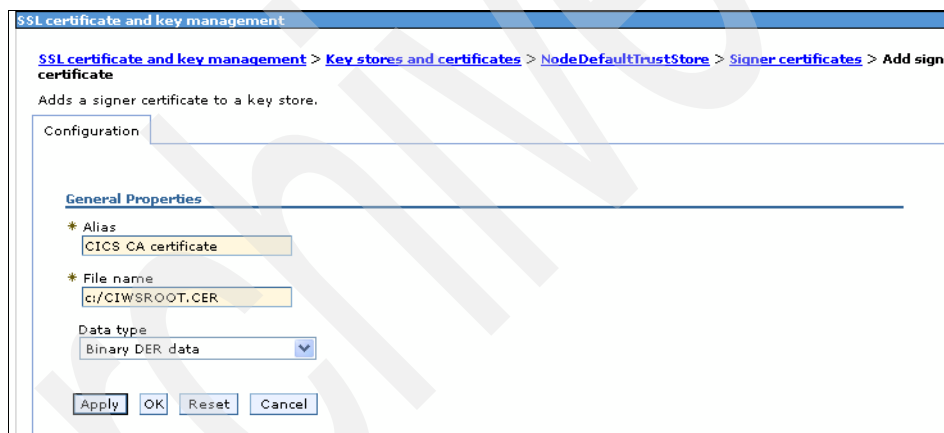


Figure 7-5 Adding CICS CA certificate to trust store

5. Click **OK** and save the changes.

Figure 7-6 shows the CICS CA certificate in the NodeDefaultTrustStore.

SSL certificate and key management > Key stores and certificates > NodeDefaultTrustStore > Signer certificates

Manages signer certificates in key stores.

Preferences

Add Delete Extract Retrieve from port

Select	Alias	Issued to	Fingerprint (SHA digest)	Expiration
<input type="checkbox"/>	default	CN=9.212.131.217, O=IBM, C=US	07:89:33:C7:88:32:A4:A3:54:25:91:DE:7B:19:56:0C:D4:D4:79:D4	Valid from 29 June 2008 to 29 June 2009.
<input type="checkbox"/>	dummyclientsigner	CN=jclient, OU=SWG, O=IBM, C=US	0B:3F:C9:E0:70:54:58:F7:FD:81:80:70:83:A6:D0:92:38:7A:54:CD	Valid from 30 July 2003 to 13 October 2021.
<input type="checkbox"/>	dummyserver signer	CN=jserver, OU=SWG, O=IBM, C=US	FB:38:FE:E6:CF:89:BA:01:67:8F:C2:30:74:84:E2:40:2C:B4:B5:65	Valid from 30 July 2003 to 13 October 2021.
<input type="checkbox"/>	cics_ca_certificate	CN=CIWSROOT, TITLE=CIWSROOT- certificate, OU=ITSO, O=IBM, L=Poughkeepsie, ST=New York, C=US	49:D1:B2:D5:E4:8A:2C:34:D0:3C:D4:04:4C:27:E1:77:32:85:1E:5A	Valid from 27 June 2008 to 01 January 2011.

Total 4

Figure 7-6 Signer certificates in trust store

7.4 Configuring the service requester

In this section we show how to configure the client J2EE application to send and receive signed SOAP messages. The steps to do this are:

1. Import the ExampleAppClientV6 application to the Application Server Toolkit (AST).
2. Configure the request generator for signing.
3. Configure the response consumer for signing.
4. Re-deploy the Web Service client application.

In the following sub-sections, we provide step-by-step details for each procedure.

Importing the base application

To configure our Web Service client to work with signed SOAP messages, we used the IBM WebSphere Application Server Toolkit V6.1 (AST). We created a new workspace and imported the ExampleAppClientV6.ear file.

Configuring the request generator for signing

The AST provides two options to configure WS-Security:

- ▶ Enabling Web services security using the WS Security wizard.

This is a feature of AST V6.1. You can:

- Add an XML digital signature
- Add XML encryption
- Add a stand alone security token

- ▶ Enabling Web services security manually using the Web Services editor.

We use the WS-Security wizard to configure the service client application for signing. Perform the following steps:

1. Expand **JSR-109 Web Services** in the Project Explorer and locate the service **DFH0XCMNService3**. This is the Web client service that places an order with the CICS catalog manager application. Right-click the **DFH0XCMNService3** service and select **Secure Web Service Client** → **Add XML Digital Signature**, as shown in Figure 7-7.

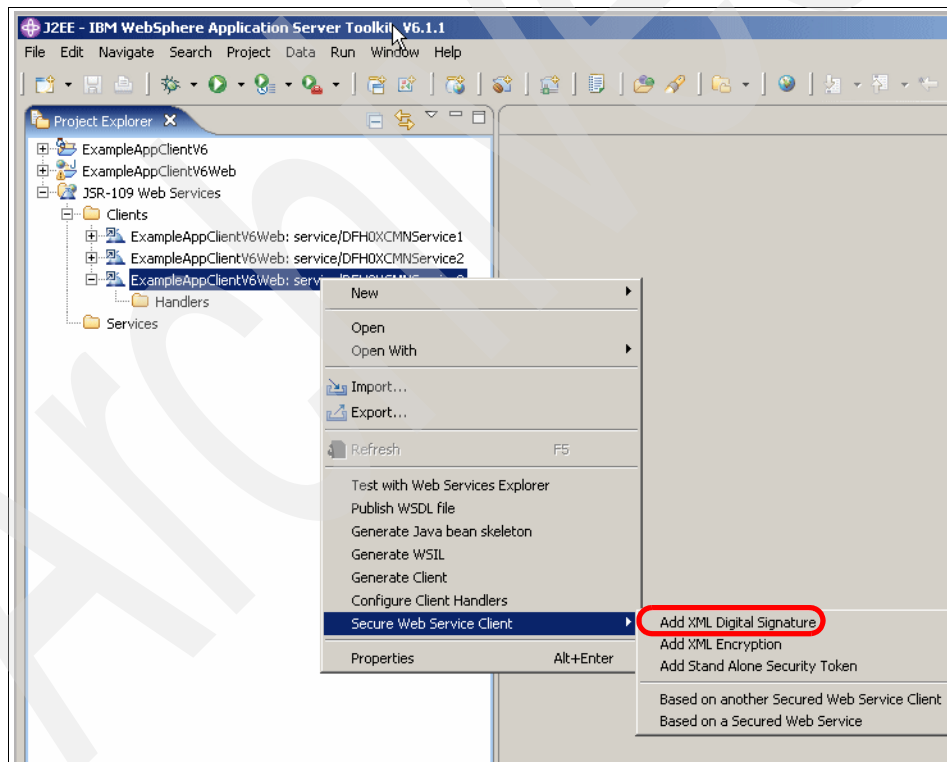


Figure 7-7 Application Server Toolkit WS-Security wizard

2. In the Client Side Request Generator page (Figure 7-8), we accept the defaults:
 - To use the RSA signature method algorithm
 - To sign the body of the SOAP message
 - To use a key information type of STRREF

WS-Security wizard - Client Side XML Digital Signature

Client Side Request Generator Digital Signature

Enter digital signature configuration for the client side request generator.

Integrity message parts

Order of validation: 1

Parts Dialect	Parts Keyword
http://www.ibm.com/websphere/webser...	body

Add Remove

Key information

Key information type: STRREF

Key locator class: com.ibm.wsspi.wsssecurity.keyinfo.KeyStoreKeyLocator

Digital signature information

Signature method algorithm: http://www.w3.org/2000/09/xmldsig#rsa-sha1

Canonicalization method algorithm: http://www.w3.org/2001/10/xml-exc-c14n#

Digest method algorithm: http://www.w3.org/2000/09/xmldsig#sha1

Transform algorithm: http://www.w3.org/2001/10/xml-exc-c14n#

< Back Next > Finish Cancel

Figure 7-8 Application Server Toolkit WS-Security wizard - page 1

We select **STRREF** (Direct reference) as the Key information type because we want WebSphere to send the public key as part of the complete certificate, since we will use the certificate for authentication in addition to decrypting the signed message.

Note: It is also possible to configure an indirect reference to the public key, such as the key identifier or the key name. In this case, CICS uses the key reference to identify the RACF certificate that contains the public key.

Click **Next** and the Token Generator page is displayed.

3. In the Token Generator page, we specify the WebSphere certificate and key store information (Figure 7-9).

Here we specify what token WebSphere should send in the request message and the key store information that allows WebSphere to sign the request (the WebSphere certificate).

WS-Security wizard - Client Side XML Digital Signature

Token Generator
Enter the token generator information.

Token generator

Token type: X509 certificate token v3

URI:

Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1

Callback handler: com.ibm.wsspi.wsssecurity.auth.callback.X509CallbackHandler

Key store information

Key store password: WebAS

Key store path: onfig\cells\Cev8-Pc7Node01Cell\nodes\Cev8-Pc7Node01\key.p12

Key store type: PKCS12

Use a key

Key alias: waswinserv01

Key password: WebAS

Use a certificate

Certificate information

X509 certificate: \$(USER_INSTALL_ROOT)\etc\ws-security\samples\intca2.cer

< Back Next > Finish Cancel

Figure 7-9 Application Server Toolkit WS-Security wizard - page 2

We accept the default Callback handler class and enter the following settings:

- **X509 certificate token v3** as the Token type

For the key store information, we select **Use a key**, and the following key store information.

- **WebAS** as Key store password
- **C:\Program Files\IBM\WebSphere\AppServer\profiles\AppSrv01\config\cells\Cev8-Pc7Node01Cell\nodes\Cev8-Pc7Node01\key.p12** as Key store path

Note: We used the default WebSphere key store. The key store path is dependent on the WebSphere install path.

- **PKCS12** as Key store type
- **waswinserv01** as Key alias
- **WebAS** as Key password

Click **Next** and the Client Side Response Consumer page is displayed.

4. In the Client Side Response Consumer page (Figure 7-10) we accept all of the defaults.

Parts Dialect	Parts Keyword
http://www.ibm.com/websphere/webser...	body

Key information
Key information type: STRREF
Key locator class: com.ibm.wsspi.wsssecurity.keyinfo.X509TokenKeyLocator

Digital signature information
Signature method algorithm: http://www.w3.org/2000/09/xmldsig#rsa-sha1
Canonicalization method algorithm: http://www.w3.org/2001/10/xml-exc-c14n#
Digest method algorithm: http://www.w3.org/2000/09/xmldsig#sha1
Transform algorithm: http://www.w3.org/2001/10/xml-exc-c14n#

Figure 7-10 Application Server Toolkit WS-Security wizard - page 3

Click **Next** and the Token Consumer page is displayed.

5. In the Token Consumer page, we specify the Token information for the response consumer (Figure 7-11).

Here we specify what token type WebSphere should expect to receive in the response message and the trust store information that allows WebSphere to verify the certificate that is used to sign the response (the CICS certificate).

The screenshot shows the 'Token Consumer' page of the 'WS-Security wizard - Client Side XML Digital Signature' dialog box. The page is titled 'Token Consumer' and contains the following fields and options:

- Token consumer:**
 - Token type: X509 certificate token v3
 - URI: (empty)
 - Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token
 - JAAAS configuration name: system.wssecurity.X509BST
- Trust options:**
 - Trust any certificate
 - Only trust certificates with the following reference:
- Key store information:**
 - Key store password: WebAS
 - Key store path: ig\cells\Cev8-Pc7Node01Cell\nodes\Cev8-Pc7Node01\trust.p12
 - Key store type: PKCS12
- Use a certificate
- Certificate information:**
 - X509 certificate: \${USER_INSTALL_ROOT}\etc/ws-security\samples\intca2.cer

At the bottom of the dialog box, there are navigation buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

Figure 7-11 Application Server Toolkit WS-Security wizard - page 4

We accept the default JAAAS configuration name and enter the following settings:

- **X509 certificate token v3** as the Token type

We select **only trust certificates with the following reference**, and the following trust store information.

- **WebAS** as Key store password
- **C:\Program Files\IBM\WebSphere\AppServer\profiles\AppSrv01\config\cells\Cev8-Pc7Node01Cell\nodes\Cev8-Pc7Node01\trust.p12** as Key store path
- **PKCS12** as Key store type

Note: In a production environment, you are likely to use CA signed certificates and a trust store. However, you can also select **Trust any certificate**. By selecting this option, the signature is validated by the certificate sent with the message without the certificate itself being validated. This option allows any client with a valid XML digital signature certificate to send signed SOAP messages to your WebSphere application server.

Click **Finish** and the WS-Security wizard creates a full set of deployment descriptor files. You can then review and edit these files using the Web Services editor by double-clicking on the **DFH0XCMNService3** service and opening the **WS Extension** and **WS Binding** tabs in the Web Deployment Descriptor.

Redeploying the Web service application

After configuring a signature security constraint for the service requester application, we exported a new EAR file, ExampleAppClientV6Signature.ear.

For further information about configuring WS-Security in WebSphere refer to *Web Services Handbook for WebSphere Application Server 6.1*, SG24-7257.

7.5 Configuring CICS for signature processing

In this section we show how to configure the CICS pipeline to receive and send signed SOAP messages, and to run the placeOrder request under the user ID associated with the WebSphere certificate contained in the request message.

We document the following steps:

1. Enabling CICS to use WS-Security support
2. Defining a new pipeline to process signed placeOrder request messages
3. Authorizing the service requester to run the placeOrder service

7.5.1 Enabling CICS to use WS-Security support

To implement signature processing, you must apply a number of updates to your CICS region. “Enabling CICS for WS-Security processing” on page 47 outlines the implementation prerequisites.

On our system, most of these prerequisites were already in place, although we did have to add the CICSTS32.CICS.SDFHWSLD library to the DFHRPL concatenation. Without this library in the DFHRPL concatenation, when testing the XML digital signature scenario, we got the message shown in Example 7-8.

Example 7-8 Error when DFHWSSE1 cannot be linked

DFHPI0500 07/09/2008 10:28:39 A6POS3C2 ORDR The CICS Pipeline Manager DFHPIPM encountered an error while trying to link to program DFHWSSE1. The program was not loadable. PIPELINE: EXSECURE.

7.5.2 Defining a new pipeline

Since the inquireCatalog and inquireSingle service requests will not be subject to XML signature processing, we need to create a new pipeline specifically for the placeOrder requests.

Creating the pipeline configuration file

Example 7-9 shows the pipeline configuration file that we used for the XML signature test scenario. It includes the user-written handler CIWSMSGO and the CICS supplied message handler DFHWSSE1. It also includes the configuration information for the DFHWSSE1 handler.

Example 7-9 Pipeline config file, ITSO_7658_wssec_signature_provider.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
  provider.xsd ">
  <service>
    <service_handler_list>
      <handler>
        <program>SNIFFER</program>
        <handler_parameter_list/>
      </handler>
      <wsse_handler>
        <dfhwsse_configuration version="1">
          <authentication trust="none" mode="signature">
            <algorithm>http://www.w3.org/2000/09/xmlsig#rsa-sha1</algorithm>
          </authentication>
          <expect_signed_body/>
          <sign_body>
            <algorithm>http://www.w3.org/2000/09/xmlsig#rsa-sha1</algorithm>
            <certificate_label>CIWSS3C2</certificate_label>
          </sign_body>
        </dfhwsse_configuration>
      </wsse_handler>
      <handler>
        <program>CIWSMSGO</program>
        <handler_parameter_list/>
      </handler>
    </service_handler_list>
  </service>
</provider_pipeline>
```

```
</service_handler_list>
</terminal_handler>
  <cics_soap_1.2_handler/>
</terminal_handler>
</service>
<apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

The CIWSMSGO message handler program changes the transaction ID to ORDS, the secure order transaction.

The `<wsse_handler>` element shown in Example 7-9 on page 244 contains a `<dfhwsse_configuration>` element that specifies configuration information for DFHWSSE1.

- ▶ In the `<authentication>` element:
 - The `trust="none"` attribute specifies that asserted identity is not used.
 - The `mode="signature"` attribute specifies that inbound messages must contain an X.509 certificate in a BinarySecurityToken.

If `<authentication mode="signature">` is specified then the client X.509 certificate must be imported to RACF and attached to the CICS key ring since CICS runs the service request under the user ID associated with the client certificate.

Note: CICS does not support certificate name filtering for signed SOAP messages.

If `<authentication mode="signature">` is not specified, and if the complete X.509 certificate is included in the request message, then the client X.509 certificate does not need to be imported to RACF since the public key required to validate the signature is extracted from the X.509 certificate contained in the SOAP message.

- ▶ `<algorithm>http://www.w3.org/2000/09/xmldsig#rsa-sha1</algorithm>` is a child element of the `<authentication>` element that specifies the URI of the signature algorithm.

Note: We found that it is necessary to specify a valid signature algorithm in service provider mode even if the algorithm used in the decryption of the signature is the one specified in the SOAP message rather than in the pipeline configuration file.

- ▶ The `<expect_signed_body/>` element indicates that the `<body>` of the inbound message must be signed. If the body of an inbound message is not correctly signed, CICS rejects the message with a security fault.
We specify `<expect_signed_body/>` because we want to prevent the placeOrder service being accessed unless the body of the SOAP message is signed.
- ▶ The `<sign_body/>` element indicates that the `<body>` of the outbound message must be signed, and provides information about how the message is to be signed.
- ▶ `<algorithm>http://www.w3.org/2000/09/xmldsig#rsa-sha1</algorithm>` is a child element of the `<sign-body>` element which specifies the URI of the signature algorithm to be used for signing the response message. This is the only signature algorithm supported by CICS for outbound messages.
- ▶ `<certificate_label>CIWSS3C2</certificate_label>` is a child element of the `<sign-body>` element that specifies the label associated with the CICS certificate installed in RACF. This certificate contains the PKDS key label of the private key that is used to sign the message response. The public key associated with the private key is then sent in the SOAP response message, allowing the signature to be validated by WebSphere.

Creating a new pipeline to process placeOrder requests

We now create the new pipeline resource definition EXSECURE and we associate this pipeline with the configuration file `ITSO_7658_wssec_signature_provider.xml` and the wsbind directory `/CIWS/S3C2/wsbind/provider/secured`.

We do this using the RDO commands in Example 7-10 and then re-installing the PIPELINE resource definition.

Example 7-10 Create pipeline for signature processing

```

CEDA COPY PIPELINE(EXPIPE01) GROUP(S3C2) AS(EXSECURE)
CEDA ALTER PIPELINE(EXSECURE) GROUP(S3C2) Configfile(/CIWS/S3C2/config/
ITSO_7658_wssec_signature_provider.xml)
Wmdir(/CIWS/S3C2/wsbind/provider/secured/)

```

We move the `placeOrder.wsbind` file to the `/CIWS/S3C2/wsbind/provider/secured/` directory so that the placeOrder Web service is dynamically installed when the EXSECURE pipeline is installed.

7.5.3 Authorizing the service requester

In this section we show the RACF commands that are used to authorize the service requester to run the placeOrder Web service.

Permitting access

In “Setting up basic security configuration” on page 161, we show the RACF commands that we used to permit the pre-defined user ID (PRIVAT01) to start the ORDR transaction.

Example 7-11 shows the RACF commands that we used to permit the service requester’s user ID (USERWS01) to start the ORDS transaction.

Example 7-11 RACF command to allow USERWS01 to run ORDS transaction

```
PERMIT CIWS3D.ORDS CLASS(TCICSTRN) ID(USERWS01) ACCESS(READ)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

Permitting surrogate access

Example 7-12 shows the RACF commands that we used to permit the pre-defined user ID (PRIVAT01) to start transactions (such as ORDS) with the user ID USERWS01.

Example 7-12 RACF commands to allow PRIVAT01 to act as surrogate for USERWS01

```
RDEFINE SURROGAT USERWS01.DFHSTART UACC(NONE) OWNER(CICRS6)
PERMIT USERWS01.DFHSTART CLASS(SURROGAT) ID(PRIVAT01) ACCESS(READ)
SETROPTS RACLIST(SURROGAT) REFRESH
```

For more information regarding surrogate user security, see *CICS Transaction Server for z/OS RACF Security Guide*, SC34-6835.

7.6 Testing the signature scenario

In order to test our XML digital signature scenario, we deploy the configured Web client application (file **ExampleAppClientV6Signature.ear**) to our WebSphere Application Server running on Windows. See “Installing and setting up the application” on page 174 for more information on deploying and testing the sample application.

We submit a placeOrder request so that a Web service request is sent from WebSphere Application Server to CICS.

In this section we show the results of testing the signature scenario, including:

- ▶ SOAP request and response messages captured with TCP/IP Monitor
- ▶ CEMT showing that the tasks run under the correct user IDs
- ▶ SOAP fault messages
- ▶ Information on configuring WebSphere TCP/IP Monitor

Request from WebSphere to CICS

An example of the signed request message sent by WebSphere is shown in Example 7-13.

Example 7-13 Signed SOAP request message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1"
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <wsse:BinarySecurityToken
        EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary"
        ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3" wsu:Id="x509bst_1"
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">MIICQCCAamgAwIBAgIERNoOdZANBgqhkiG9w0BAQUFADBZMQswCQYDVQQGEwJVUzEJMAcGA1UEERMAQkwBwYDVQIEwAxCTAHBgNVBACTADEMMaGA1UEChMDSUJNMQkwBwYDVQQLEwAxEDA0BgNVBAMTB3dhc2N1cnQwHhcNMdYwDA5MTkxNjA3W
hcNMDcwODA5MTkxNjA3WjBZMQswCQYDVQQGEwJVUzEJMAcGA1UEERMAQkwBwYDVQIEwAxCTAHBgNVBACTADEMMaGA1UEChMDSUJNMQkwBwYDVQQLEwAxEDA0BgNVBAMTB3dhc2N1cnQwZ8wDQYJKoZIhvcNAQEBBQADgYOAQIBGJAoGBAIEtFDoacSSCWtqNYVPDd3yMGMIq4GpSN
UVajzLdW+h1NIgaKzZfhi0o6BJxQcD+Ty+pKRS1bPLYG9B+9LGo+06dNJmAXDVGNIqSgkNY
x112oWRBCJciU5nBIB3a+TU0e2wYEak+rJ3Mb1B/TjA3ottykjft0yjRoh197wt65j/AgMB
AAGjFTATMBEGA1UdDgQKBAhJYPbuSs76STANBgqhkiG9w0BAQUFAA0BgQB1SEGGNW4ruu
a80hItDXERBA3560nzLw05n+eb2JdZu0i1yHNGfp8+k4b+9F9Dj0PzGEJzjgspqMTraHK0J
P1co7yIG/RUuX+gicGN+dI2A8frLsIS2IUMB66FqQR/ua0YjGJCuYfzKm2Cc0PUFTQYqMMk
nuj+DBTcAoDp+GP4Q==
      </wsse:BinarySecurityToken>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
          <ds:CanonicalizationMethodAlgorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
            <ec:InclusiveNamespaces PrefixList="xsi xsd soapenv soapenc wsse
              ds" xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" />
          </ds:CanonicalizationMethod>
          <ds:SignatureMethod Algorithm=
            "http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
        </ds:Signature>
      </ds:Signature>
    </wsse:Security>
  </soapenv:Header>
  </soapenv:Envelope>
```



```

<ds:Reference URI="#wssecurity_signature_id_0">
  <ds:Transforms>
    <ds:Transform
      Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      <ec:InclusiveNamespaces PrefixList="xsi xsd soapenv soapenc wsu
        p635" xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" />
    </ds:Transform>
  </ds:Transforms>
  <ds:DigestMethod Algorithm=
    "http://www.w3.org/2000/09/xmlsig#sha1" />
  <ds:DigestValue>fh93yxjxoa8CwZ9R0qH2m1eWdY0=</ds:DigestValue>
</ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>I53NN8zcbfRmnz3LwVf3vZeFFDpp2WAKgBU9+Sfu/I71G3Wh8tsh
  3fRngwt7qLmRH3qGzjpsMWCbxKssyOP+drSyBwvUIz4FukxyJ5pASVot7qPNRfj9p60/Y
  0Tu26z0W9GQSc1jFqRp16+tNZqHwLiMfpmQyJggIF8izQU8nkQ=
</ds:SignatureValue>
<ds:KeyInfo>
  <wsse:SecurityTokenReference>
    <wsse:Reference URI="#x509bst_1"
      ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-ws
        s-x509-token-profile-1.0#X509v3" />
    </wsse:SecurityTokenReference>
  </ds:KeyInfo>
</ds:Signature>
</wsse:Security>
</soapenv:Header>
<soapenv:Body wsu:Id="wssecurity_signature_id_0"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecuri
    ty-utility-1.0.xsd">
  <p635:DFHOXCMN xmlns:p635="http://www.DFHOXCMN.DFHOXCP5.Request.com">
    <p635:ca_request_id>010RDR</p635:ca_request_id>
    <p635:ca_return_code>0</p635:ca_return_code>
    <p635:ca_response_message>[C@752a752a</p635:ca_response_message>
    <p635:ca_order_request>
      <p635:ca_user ID>nigel</p635:ca_user ID>
      <p635:ca_charge_dept>itso</p635:ca_charge_dept>
      <p635:ca_item_ref_number>10</p635:ca_item_ref_number>
      <p635:ca_quantity_req>1</p635:ca_quantity_req>
      <p635:filler1 xsi:nil="true" />
    </p635:ca_order_request>
  </p635:DFHOXCMN>
</soapenv:Body>
</soapenv:Envelope>

```

- ▶ The header includes the `mustUnderstand="1"` attribute, which indicates that either this header must be processed or a SOAP fault thrown.
- ▶ The `<wsse:BinarySecurityToken>` contains the base64binary encoding of the WebSphere certificate. This includes the public key that CICS uses to verify the signature.
- ▶ The `<ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />` element specifies the algorithm used to sign the message digest.
- ▶ The `<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />` element specifies the algorithm used to produce the message digest.
- ▶ The `<ds:DigestValue>` element contains the value of the message digest.
- ▶ The `<ds:SignatureValue>` element contains the value of the signed message digest. It is the digest value encrypted with the requester's private key.

CEMT INQUIRE TASK

Figure 7-12 shows the ORDS transaction running with the user ID `USERWS01` (the user ID associated with the WebSphere certificate), while the `ORDR` transaction runs with the `PRIVAT01` user ID.

```

INQUIRE TASK
STATUS: RESULTS - OVERTYPE TO MODIFY
Tas(0000055) Tra(ORDR)          Sus Tas Pri( 001 )
  Sta(U ) Use(PRIVAT01 ) Uow(BF3C2E9E2114284E) Hty(RZCBNOTI)
Tas(0000056) Tra(ORDS)          Sus Tas Pri( 001 )
  Sta(U ) Use(USERWS01 ) Uow(BF3C2E9E24801B8B) Hty(EDF      )
Tas(0000058) Tra(CEDF) Fac(E024) Sus Ter Pri( 001 )
  Sta(SD) Use(CICRSR6 ) Uow(BF3C2E9E28D4A149) Hty(ZCIOWAIT)
Tas(0000059) Tra(CEMT) Fac(E025) Run Ter Pri( 255 )
  Sta(T0) Use(CICRSR6 ) Uow(BF3C2EA64198CCCE)

SYSID=S3C2 APPLID=A6POS3C2

```

Figure 7-12 ORDS executing with user ID associated with WebSphere certificate

Response from CICS to WebSphere

The corresponding signed response message sent by CICS is shown in Example 7-14.

Example 7-14 Signed SOAP response message

```
<?xml version="1.0" encoding="UTF8" standalone="no" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd" SOAP-ENV:mustUnderstand="1"
      xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <wsse:BinarySecurityToken
        EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary"
        ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"
        wsu:Id="x509cert00">MIIC4DCCAkmgAwIBAgIBADANBgkqhkiG9w0BAQUFADB8MQswC
        QYDVQQGEwJVUzERMA8GA1UECBMITkVXIFlPUksxETAPBgNVBACICEVORE1DT1RUMQOW
        CwYDVQQKEwRJVFNPMQOWCwYDVQLLEwRQU1NDMRYwFAYDVQQMEw1NQURFIFVQIFRJVEx
        FMREwDwYDVQQDEWhDSUNTQ0VSVD AeFw0wNTAxMDEwNDAwMDBaFw0xNTAxMDEwMzU5NT
        1aMHwx CzAJBgNVBAYTA1VTMREwDwYDVQQIEWhORVcgWU9SSzERMA8GA1UEBxMIRU5ES
        UNPVFQxDALBgNVBAoTBE1U08xDALBgNVBAsTBFBTUOMx FjAUBgNVBAwTUDU1BREUg
        VVAgVE1UTEUxETAPBgNVBAMTCENJQ1NDRVJUMIGFMAOGCSqGSIb3DQEBAQUAA4GNAD
        CiQKBgQC5EHPavGQtIkVbP0+qNaBFF79tNk3aXDur3Pup4KIycA7JueLtm6sLOyQDNF
        snZgw811W97EUKIsT55jwYHZcGSR2TjNxs wdGrt4rt8EPgwtN3WS1609uWrhVTug4Uu
        MEOXpivdcNDTwRbQgvMtXdOK0WvdSmrbEwBiB4LSdm2pQIDAQABo3IwcDA/Bg1ghkgB
        hvhCAQOEMhMwR2VuZXJhdGVkIGJ5IHRoZSBTZWN1cm10eSBTZXJ2ZXIgzM9yIHovT1M
        gKFJBQ0YpMA4GA1UdDwEB/wQEAWIEUDAdBgNVHQ4EFgQU6ztx+OkFbSysc93LQBZkSh
        ymaPgWdQYJKoZIhvcNAQEFBQADgYEATtVMjr2pTz47TF92RoTpUneZxq2eMz7LJJSLD
        u37ya+qCLaxbTPB7XfKggq+egYyYXDYi4mIsfpRq3MHBM/nFivWtPONpcMcEz8hfH1q
        Gp3mze+NfuRV4iLpXtHJR9rDPTQsvgwpuIrWfeMX+/IrxYXtY391h4L78G8s/CQj23U
        =
      </wsse:BinarySecurityToken>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
          xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
          xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
          <ds:CanonicalizationMethod
            Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
            <c14n:InclusiveNamespaces
              xmlns:c14n="http://www.w3.org/2001/10/xml-exc-c14n#"
              PrefixList="ds wsu xenc SOAP-ENV soapenc soapenv xsd xsi "/>
            </c14n:InclusiveNamespaces>
          </ds:CanonicalizationMethod>
        </ds:SignedInfo>
      </ds:Signature>
    </wsse:Security>
  </SOAP-ENV:Header>
</SOAP-ENV:Envelope>
```

```

</ds:CanonicalizationMethod>
<ds:SignatureMethod
  Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
<ds:Reference URI="#TheBody">
  <ds:Transforms>
    <ds:Transform
      Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"
      <c14n:InclusiveNamespaces
        xmlns:c14n="http://www.w3.org/2001/10/xml-exc-c14n#"
        PrefixList="wsu SOAP-ENV soapenc soapenv xsd xsi "/>
      </ds:Transform>
    </ds:Transforms>
    <ds:DigestMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <ds:DigestValue>tLbsd1SPsgrzGZ5bD0dtyRoHDW0=</ds:DigestValue>
  </ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>Zyxxxq/n7yDaGDYwsIIS21MFbDdMWNruFJ/tT5HuWi0Db6N7kS
DFccM27mQb1uEVFjkNjkKz0n0LNWgIzGqoBU4cV3hu6V0Kr4Qg8CnwL06yDpyQYYC/e
5rcjfCQUycgJ1JVkdqd+ERN9hwbXX3wZZX3PVZSH5Qs0mH/aJ0eSME=
</ds:SignatureValue>
<ds:KeyInfo>
  <wsse:SecurityTokenReference><wsse:Reference URI="#x509cert00"
    ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
    wss-x509-token-profile-1.0#X509v3"/>
  </wsse:SecurityTokenReference>
</ds:KeyInfo>
</ds:Signature>
</wsse:Security>
</SOAP-ENV:Header>
<SOAP-ENV:Body
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
  wssecurity-utility-1.0.xsd" wsu:Id="TheBody">
  <DFHOXCMNResponse
    xmlns="http://www.DFHOXCMN.DFHOXCP5.Response.com">
    <ca_request_id>010RDR</ca_request_id>
    <ca_return_code>0</ca_return_code>
    <ca_response_message>ORDER SUCESSFULLY PLACED
    </ca_response_message>
    <ca_order_request>
      <ca_user ID>nigel</ca_user ID>
      <ca_charge_dept>itso </ca_charge_dept>
      <ca_item_ref_number>10</ca_item_ref_number>
      <ca_quantity_req>1</ca_quantity_req>
      <filler1>...</filler1>
    </ca_order_request>
    </DFHOXCMNResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

The `<wss:BinarySecurityToken>` contains the base64binary encoding of the CICS certificate. This includes the public key that WebSphere uses to verify the signature. The rest of the SOAP headers in the response message are similar to those in the request message.

7.6.1 SOAP fault messages

In this section we show some of the SOAP faults that you might see when testing signature processing.

- ▶ CICS expects a signed message but the service requester does not sign the request.

The SOAP fault for this message, shown in Example 7-15, highlights that CICS is expecting a BinarySecurityToken, which is not sent by WebSphere.

Example 7-15 DFHWSSE1 SOAP fault - requester certificate not sent

```
<SOAP-ENV:Body>
  <SOAP-ENV:Fault>
    <faultcode>wss:InvalidSecurity</faultcode>
    <faultstring>ERROR: Caught *XSECException* during operation:
      processMessage()</faultstring>
    <detail>
      <e:myfaultdetails xmlns:e="http://www.ibm.com/software/htp/cics/wssec">
        <message>SecurityContext::processCredentials - Expected
          BinarySecurityToken does not exist</message>
        <errorcode>1</errorcode>
      </e:myfaultdetails>
    </detail>
  </SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

- ▶ The requester's certificate is not associated to a user ID.

The SOAP fault for this message, shown in Example 7-16, highlights that there is no RACF user ID defined for the WebSphere certificate.

Example 7-16 DFHWSSE1 SOAP fault - requester certificate not associated with user ID

```
<SOAP-ENV:Body>
  <SOAP-ENV:Fault>
    <faultcode>wss:FailedAuthentication</faultcode>
    <faultstring>ERROR: Caught *XSECException* during operation:
      processMessage()</faultstring>
    <detail>
      <e:myfaultdetails xmlns:e="http://www.ibm.com/software/htp/cics/wssec">
```

```
<message>XSECKeyInfoResolverZos::extractuser ID - Either no RACF user
  ID is defined for this certificate, or the certificate status is
  NOTRUST.</message>
  <errorcode>1</errorcode>
</e:myfaultdetails>
</detail>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

Note: The SOAP faults shown above were captured for a CICS TS V3.1 system. The format of SOAP fault messages changed with CICS TS V3.2. The SOAP faults shown below were captured for a CICS TS V3.2 system.

- ▶ CICS cannot sign the response message because the certificate specified in the <certificate_label> is not in the keyring.

The SOAP fault for this message, shown in Example 7-17, highlights that certificate cannot be retrieved.

Example 7-17 DFHWSSE1 SOAP fault - signing certificate not in keyring

```
<SOAP-ENV:Body>
  <SOAP-ENV:Fault>
    <faultcode>wsse:SecurityTokenUnavailable</faultcode>
    <faultstring>Referenced security token could not be retrieved
  </faultstring>
  </SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

- ▶ CICS cannot sign the response message because the certificate specified in the <certificate_label> has not been created with the ICSF option.

The SOAP fault for this message, shown in Example 7-18, highlights that certificate cannot be retrieved.

Example 7-18 DFHWSSE1 SOAP fault - signing certificate not created with ICSF

```
<SOAP-ENV:Body>
  <SOAP-ENV:Fault>
    <faultcode>wsse:InvalidSecurityToken</faultcode>
    <faultstring>An invalid security token was provided</faultstring>
  </SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

7.6.2 WebSphere Application Server TCP/IP Monitor

WebSphere Application Server provides a tool called `tcpmon`, which works like a proxy server, passing TCP/IP requests on to another server and directing the returned responses. It is a useful tool for monitoring SOAP request and response messages as they pass between service requester and service provider. The function of this tool is the same as the TCP/IP Monitor which is provided with the AST.

We used `tcpmon` to monitor signed SOAP messages. To execute the monitor, we use the command shown in Example 7-19 on the Windows machine that is running WebSphere Application Server.

Example 7-19 Starting `tcpmon`

```
"C:\Program Files\IBM\WebSphere\AppServer\java\jre\bin\java.exe" -cp  
"C:\Program  
Files\IBM\WebSphere\AppServer\runtimes\com.ibm.ws.admin.client_6.1.0.jar"  
com.ibm.ws.webservices.engine.utils.tcpmon
```

This starts `tcpmon`, as shown in Figure 7-13.

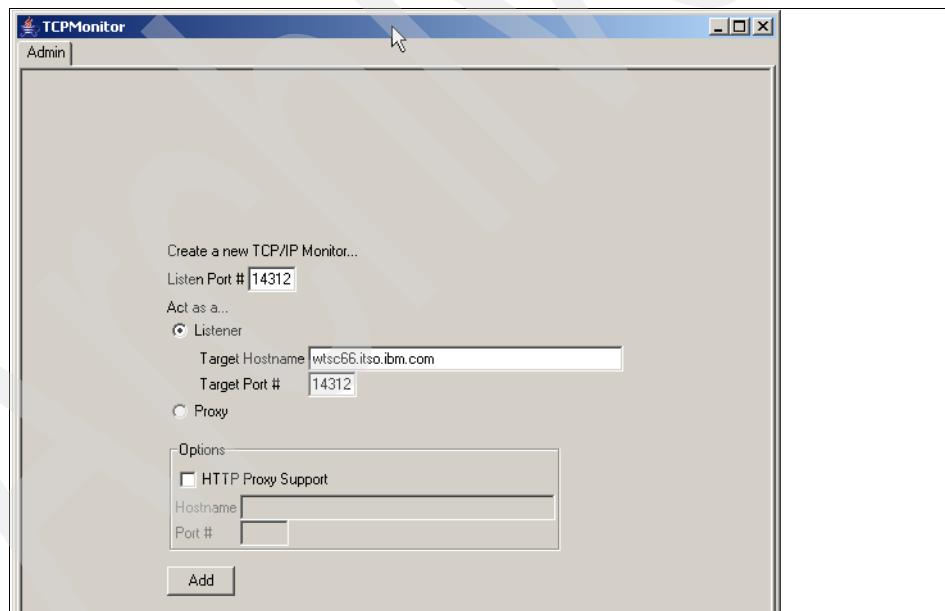


Figure 7-13 `tcpmon` configuration

Figure 7-13 shows how we configured `tcpmon` to listen on port 14312, and to forward requests to the target hostname `wtsc66.itso.ibm.com` and port 14312.

Figure 7-14 shows how we then configured the service endpoints for the Catalog example application so that requests are sent to tcpmon.

The screenshot displays the 'CICS Example - Catalog Application' configuration window. It features a sidebar with three menu items: 'LIST ITEMS', 'INQUIRE', and 'ORDER ITEM'. The main area is titled 'Configure Application' and contains three sections for configuring service endpoints:

- Inquire Catalog Service Endpoint:** Both 'Current' and 'New' fields are set to `http://localhost:14312/exampleApp/inquireCatalog`.
- Inquire Item Service Endpoint:** Both 'Current' and 'New' fields are set to `http://localhost:14312/exampleApp/inquireSingle`.
- Place Order Service Endpoint:** Both 'Current' and 'New' fields are set to `http://localhost:14312/exampleApp/placeOrder`.

At the bottom of the configuration area, there are two buttons: 'SUBMIT' and 'RESET'.

Figure 7-14 Configure Catalog application for tcpmon

Tcpmon captures TCP/IP requests and provides functions to save, load, and resend monitored SOAP requests. Furthermore, for the request and response message, an XML format can be applied, which makes a SOAP message much easier to read.

Figure 7-15 shows the tcpmon window for a sample placeOrder SOAP request.

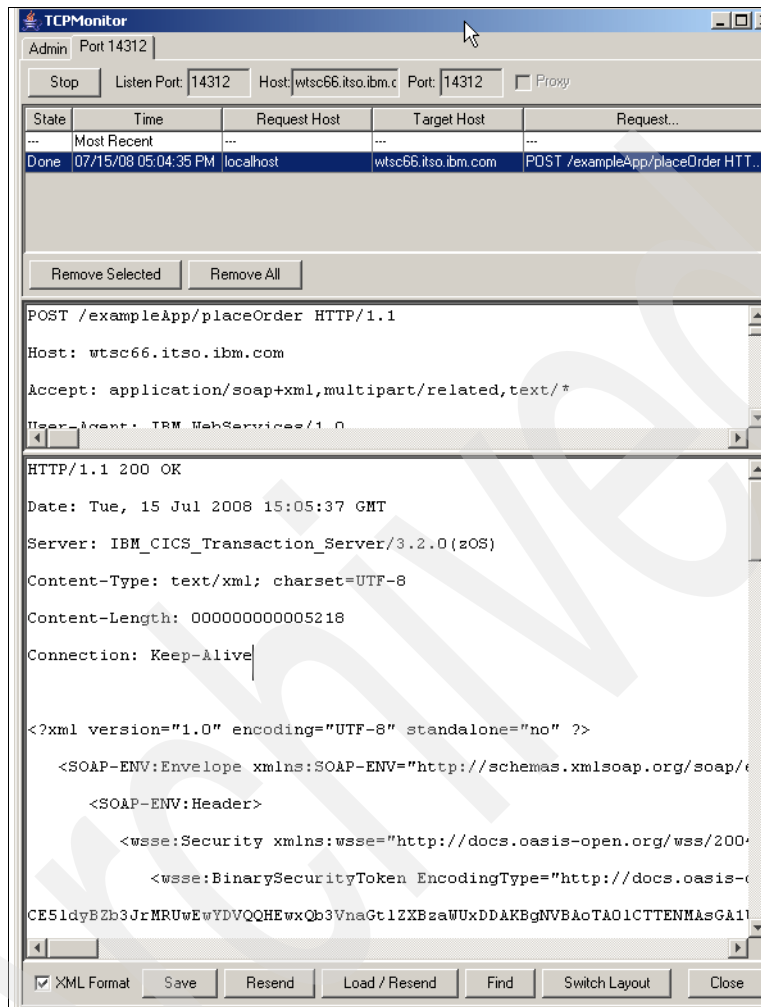


Figure 7-15 Sample SOAP request in tcpmon

Tcpmon is a useful tool for validating that SOAP messages contain the appropriate security tokens and signatures.

Archived

Identity assertion with WebSphere for z/OS

Identity assertion is an authentication mechanism that is applied among three parties: a client, an intermediary server, and a target server. The client authenticates with the intermediary server and the intermediary server asserts the client's identity to the target server.

In this chapter we show how a client can be authenticated by WebSphere Application Server for z/OS and how the client's identity can be asserted to CICS when the J2EE application running in WebSphere makes a Web service call to CICS.

We provide step-by-step security configurations for configuring the CICS service provider pipeline and the service requester J2EE application using the WebSphere Application Server Toolkit.

8.1 Scenario overview

In this chapter we show how WebSphere Application Server can assert an identity to CICS, thus avoiding the need to authenticate the user's credentials multiple times. This scenario is particularly relevant when WebSphere Application Server is running on z/OS and sharing the same RACF user registry as CICS.

Identity assertion is an extended security mechanism supported by CICS TS V3 and WebSphere Application Server V6. WebSphere Application Server must establish a trust relationship with the CICS region by authenticating itself and then by being recognized as a trusted partner of the CICS region. This can be done using one of two different models:

Trust token	WebSphere Application Server sends a trust token to CICS.
Blind trust	Trust is established at the transport level rather than at the SOAP message level.

8.1.1 Blind trust model

Support for the blind trust model was introduced in CICS TS V3.2. In this scenario we use the blind trust model which offers significant performance advantages over the trust token model.

The user identity is transported between WebSphere Application Server and CICS within a *UsernameToken* element.

Important: When the blind trust model is used, a trust relationship between WebSphere Application Server and CICS should be established. This can be done using a transport-based mechanism such as SSL client authentication. See Chapter 6., “Enabling SSL” on page 181 for information on how to configure SSL client authentication between WebSphere Application Server and CICS.

We show how the identity passed in the SOAP message is used as the user ID under which the CICS pipeline transaction runs. This scenario is shown in Figure 8-1.

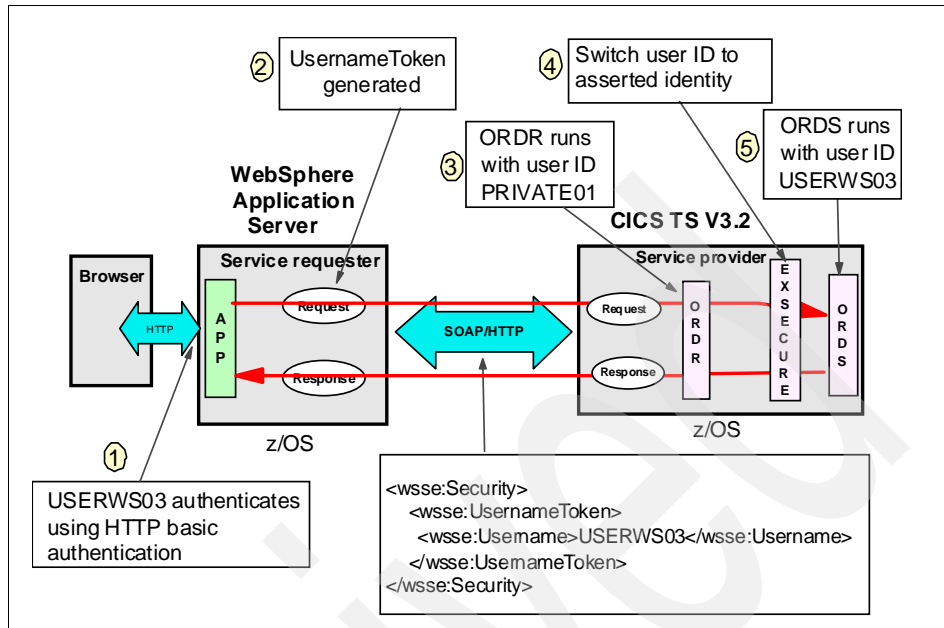


Figure 8-1 Identity assertion scenario with WebSphere on z/OS

The sequence of steps is as follows:

1. The J2EE application (in our case the Catalog manager application) is configured with a security constraint so that users must authenticate and the application is executed with the caller's identity (RunAS caller). The client USERWS03 authenticates with WebSphere Application Server using HTTP basic authentication.
2. The J2EE application makes a Web service request to CICS. The Web service requester is configured for identity assertion so that a UsernameToken element consisting of a Username only is sent by WebSphere Application Server in the WS-Security SOAP header.
3. The CICS region receives the SOAP Request and looks up the matching URIMAP. The URIMAP instructs CICS that requests for the path /exampleApp/placeOrder are to use the WEBSERVICE placeOrder, the PIPELINE EXSECURE, the transaction ORDR and the user ID PRIVATE01. CICS starts pipeline processing using this information.
4. The EXSECURE pipeline includes the CICS-supplied security handler which is configured for identity assertion with blind trust and an authentication mode of basic so that CICS puts the asserted user ID USERWS03 in the DFHWS-USERID container.

The pipeline also includes the user-written message handler CIWSMSGO which sets the context container DFHWS-TRANID to contain ORDS for any placeOrder request.

5. The ORDS transaction runs under the caller's user ID USERWS03 and all further resource security checks for this request (for example, program and database security) are then performed against the USERWS03 user ID.

8.1.2 Single sign-on

Identity assertion enables a form of single sign-on (SSO) support, in which Web users authenticate once when accessing WebSphere Application Server and then the user's identity is propagated to other servers that are accessed by the WebSphere application.

In this scenario, WebSphere Application Server for z/OS behaves in a way that traditional z/OS address spaces behave. Once the user has been authenticated, the user ID flows with any work done within the z/OS system. This is similar to the way that MRO requests in CICS are managed, for example..

Note: In Chapter 10, "Enabling WS-Trust with TFIM" on page 313 we show an alternative form of single sign-on support using the Lightweight Third Party Authentication (LTPA) protocol.

8.2 Preparation

In this section we provide a summary of the software used in the scenario and the main definitions used for the CICS and WebSphere servers.

8.2.1 Software checklist

The software we used is listed in Table 8-1.

Table 8-1 Software used in the Id assertion scenario

Windows	z/OS
Internet Explorer V7.0	z/OS V1.9
Windows XP Professional Version 2002 SP3	CICS Transaction Server V3.2
IBM WebSphere Application Server Toolkit V6.1.1.6	IBM WebSphere Application Server V6.1.0.17

Windows	z/OS
	Our J2EE applications <ul style="list-style-type: none"> ▶ ExampleAppClientV6.ear The Web client for the CICS catalog example application supplied with CICS TS.
	Our user-written CICS message handler programs <ul style="list-style-type: none"> ▶ CIWSMSGO This program changes the transaction ID for placeOrder requests to ORDS. ▶ SNIFFER (message handler program) This program browses through the containers available in the pipeline.

8.2.2 Definition checklist

The definitions we used are listed in Table 8-2.

Table 8-2 Settings used in the XML digital signature scenario

Value	CICS TS	WebSphere Application Server
IP name	wtsc66.itso.ibm.com	mve2.pssc.ibm.com
TCP/IP port	14313	8148
Jobname	CIWSS3C3	W1SR12
APPLID	A6POS3C3	
TCPIP SERVICE	S3C3	
Provider PIPELINES	EXPIPE01 for inquireSingle and inquireCatalog services EXSECURE for placeOrder service	
Configuration files	ITSO_7658_basicsoap12provider.xml for inquireSingle and inquireCatalog services ITSO_7658_wssec_idassertionUser name_provider.xml for placeOrder service	

The user IDs we used in our configuration are listed in Table 8-3.

Table 8-3 User IDs used in the XML digital signature scenario

Value	CICS TS
CICS region user ID	CIWS3D
User ID for which we wish to permit access to inquireSingle and inquireCatalog services	PUBLIC01
User ID for which we wish to permit access to the ORDR transaction of the placeOrder service	PRIVAT01
User ID for which we wish to permit access to the ORDS transaction of the placeOrder service	USERWS03

Because the Catalog application inquireSingle and inquireCatalog services are read-only services, whereas the placeOrder service updates the database, we show here how to secure the placeOrder service.

We run the inquireSingle and inquireCatalog services under a predefined user ID PUBLIC01 which is defined in inquireSingle and inquireCatalog wsbind files.

We document the following procedures to enable support for identity assertion with blind trust:

- ▶ Configuring the WebSphere service requester to send the UsernameToken
- ▶ Configuring CICS identity assertion
- ▶ Testing the identity assertion scenario

8.3 Configuring the service requester

In this section we show how to configure the client J2EE application to send a UsernameToken. The steps to do this are:

- ▶ Import the ExampleAppClientV6 application to the Application Server Toolkit (AST).
- ▶ Configure the request generator.
- ▶ Add a security constraint
- ▶ Re-deploy the Web Service client application.

In the following sub-sections we provide step-by-step details for each procedure.

Importing the base application

To configure our Web Service client we used the IBM WebSphere Application Server Toolkit V6.1 (AST). We created a new workspace and imported the ExampleAppClientV6.ear file.

Configuring the request generator for identity assertion

The AST provides two options to configure WS-Security:

- ▶ Enabling Web services security using the WS Security wizard.
We use this option in “Configuring the request generator for signing” on page 237.
- ▶ Enabling Web services security manually
If the WS Security wizard does not meet your needs, you have the option of manually securing Web services. This process gives you more control over the security settings.
We use this option with the identity assertion scenario because the WS Security wizard does not provide support for identity assertion.

Perform the following steps:

1. Expand **JSR-109 Web Services** in the Project Explorer and locate the service **DFH0XCMNService3**. This is the Web client service that places an order with the CICS catalog manager application.
Double click the **DFH0XCMNService3** service and the Web Deployment Descriptor is displayed. Select the **WS Extension** page which is used for editing the client's deployment extension file (ibm-webservicesclient-ext.xmi).
Figure 8-2 shows the WS Extension page in the Deployment Descriptor Editor.

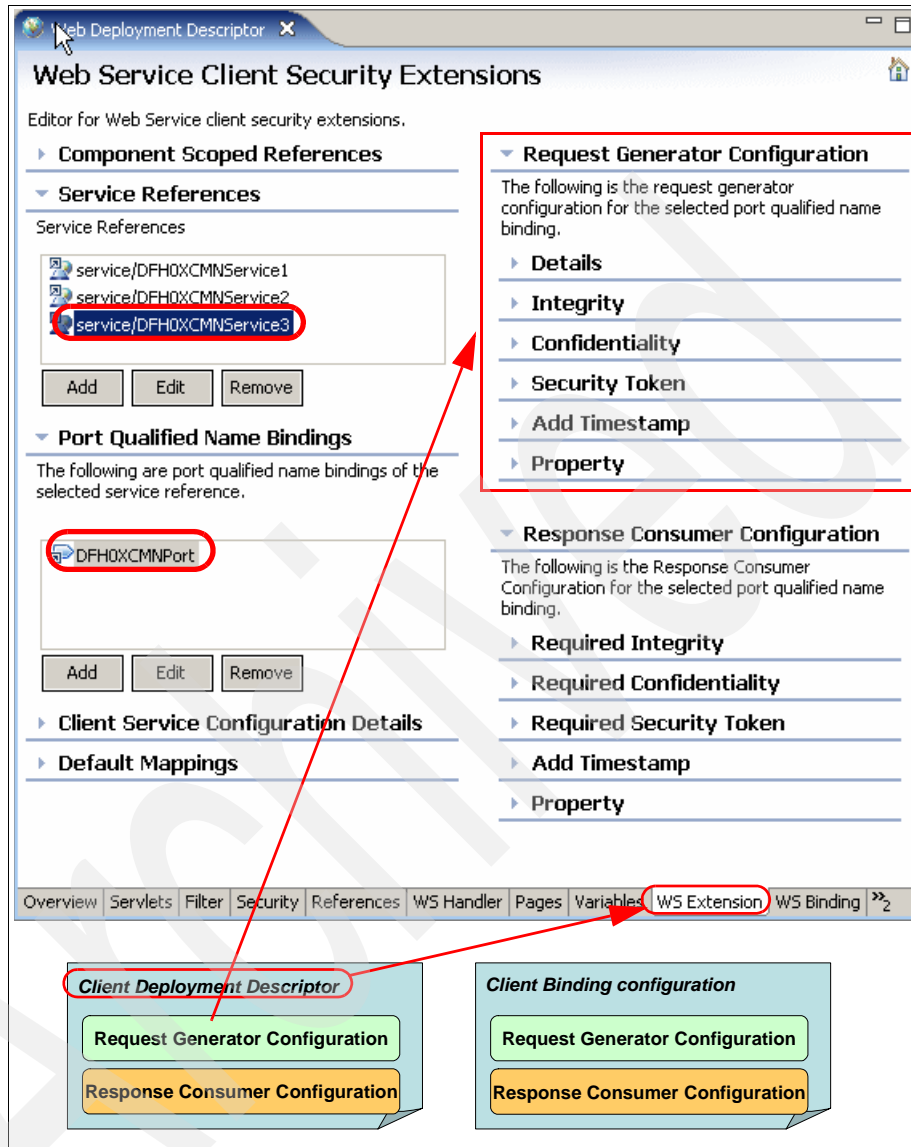


Figure 8-2 WS Extension page in Web service client Deployment Descriptor Editor

2. Click the **service/DFH0XCMNService3** reference (for the placeOrder service) and **DFH0XCMNPort** (the port binding for the service reference).
3. In the Request Generator Configuration, expand **Security Token** and click **Add**. Enter the name assertedId and select the **Username Token** token type

from the drop-down list. When you select a Token type, the Local name is filled in automatically (Figure 8-3). We left the URI empty.

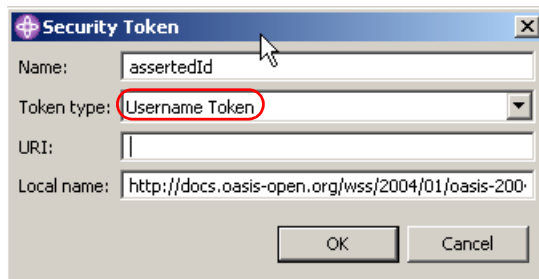


Figure 8-3 UsernameToken

4. Clicked **OK** and a security token is created. Save the configuration by pressing Ctrl+s.
5. After specifying the security token, a corresponding token generator must be specified in the binding configuration. The WS Binding page is for editing the client's binding file, so you can specify how to apply the required security.

Click the **WS Binding** tab. Figure 8-4 shows the WS Binding page in the Deployment Descriptor Editor.

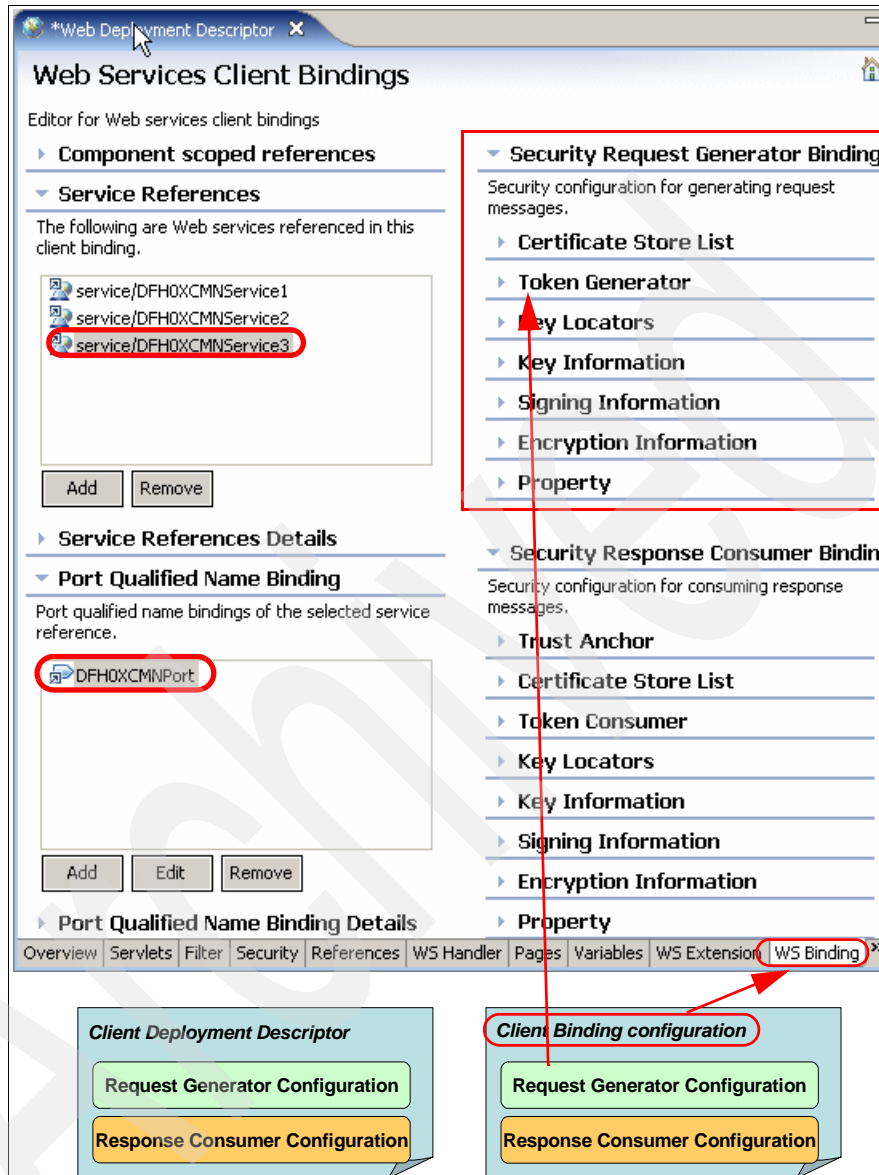


Figure 8-4 WS Binding page in the Deployment Descriptor Editor

6. Select **Token Generator** and click **Add**. Enter a Token generator name assertedIdToken and allow the Token generator class to default as com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator

7. To select the security token, expand the drop-down list and select the **assertedId** security token that was defined previously on the WS Extension page.
8. Check **Use value type** and select the **Username Token** value type from the drop-down list. This selection fills the local name and callback handler.
9. Do not enter a user ID or password as we want WebSphere to assert the caller's identity.

Instead, add the following Callback Handler properties and set the value of each one to **true**.

- `com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed`
This property specifies that identity assertion is to be used.
- `com.ibm.wsspi.wssecurity.token.IDAssertion.useRunAsIdentity`
This property specifies that the caller's identity (RunAs identity) is to be set as the user ID in the UsernameToken.

Figure 8-5 shows the configured Token Generator.

Token Generator

Token generator name: `assertedIdtoken`

Token generator class: `com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator`

Security token: `assertedId`

Use value type

Value type: `Username Token`

Local name: `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken`

URI:

Callback handler: `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`

User ID:

Password:

Use key store

Password:

Path:

Type:

Key:

Alias:	Key password:	Key name:

Add Remove

Callback Handler Property:

Name:	Value:
<code>com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed</code>	<code>true</code>
<code>com.ibm.wsspi.wssecurity.token.IDAssertion.useRunAsIden...</code>	<code>true</code>

Add Remove

Figure 8-5 Token Generator

10. Click **OK**, and the token generator is created. Save the configuration.

Adding a Web security constraint

We add a Web security constraint to the Catalog Web application. If WebSphere application security is enabled, and a Web security constraint is set for a particular resource, the resource is secured and users of the application must authenticate themselves.

Within the application EAR file, the Web application archive ExampleAppClientV6 contains file WEB-INF/web.xml. We added the security constraint as shown in Example 8-1.

Example 8-1 Web application deployment descriptor security constraint

```
<security-constraint>
  <display-name>
    CatalogSecurityConstraint</display-name>
  <web-resource-collection>
    <web-resource-name>CatalogResource</web-resource-name>
    <url-pattern>/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>PUT</http-method>
    <http-method>HEAD</http-method>
    <http-method>TRACE</http-method>
    <http-method>POST</http-method>
    <http-method>DELETE</http-method>
    <http-method>OPTIONS</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>
      Role required to run Catalog application</description>
    <role-name>CATALOG</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
<security-role>
  <description>
    Role to run Catalog application</description>
  <role-name>CATALOG</role-name>
</security-role>
</security-role>
```

</security-role>

This security constraint specifies that the Web application user must:

- ▶ Login to the WebSphere application server using basic authentication (`<auth-method>BASIC</auth-method>`)
- ▶ Be authorized to role CATALOG (`<auth-constraint>` with `<role-name>CATALOG</role-name>`)

Setting RunAs

A RunAs setting on a servlet or EJB™ affects the Java thread identity of methods invoked on a subsequent call. WebSphere Application Server RunAs policy allows three choices in assigning the Java thread identity for the current request:

- ▶ *Caller* uses the caller's identity for the method selected and to propagate it to any subsequent methods invoked or J2EE resources accessed. This is the default behavior.
- ▶ *Server* indicates that the method should assume the identity of the WebSphere servant region.
- ▶ *Role* the application assembler selects the name of a security role. Authorization is performed by checking that the principal name has been assigned to one of the required security roles.

In our tests, we use the default RunAs setting (Caller).

Defining a security role

Security roles are used to control access to J2EE resources such as servlets and EJBs. The name of the roles are defined in the application deployment descriptor, for example, in Example 8-1 on page 270 we defined a role-name CATALOG.

There are two basic ways to control role access with WebSphere Application Server for z/OS:

- ▶ Using SAF EJBROLE profiles
- ▶ Using WebSphere bindings

We defined the EJBROLE **W1.CATALOG** and authorized the user ID USERWS03 to the EJBROLE.

Note: W1 is the security domain prefix for our WebSphere Application Server. The specification of a security domain prefix affects the specific EJBROLE profiles used by WebSphere Application Server for z/OS. This enables you to deploy the same application on different cells, but have different user to role mappings if desired.

We use RACF to create the EJBROLE class and authorize the USERWS03 user ID to the role using the commands shown in Example 8-2:

Example 8-2 Create EJBROLE and permit access to USERWS03

```
RDEFINE EJBROLE W1.CATALOG UACC(NONE)
PERMIT W1.CATALOG CLASS(EJBROLE) ID(USERWS03) ACCESS(READ)
SETROPTS RACLIST(EJBROLE) REFRESH
```

Redeploying the Web service application

After configuring a Web security constraint for the service requester application, we exported a new EAR file called ExampleAppClientV6IdAssertionuserIDSec.ear.

For further information about configuring WS-Security in WebSphere refer to *Web Services Handbook for WebSphere Application Server 6.1*, SG24-7257.

8.4 Configuring CICS for identity assertion

In this section we show how to configure the CICS pipeline to receive an asserted identity, and to run the placeOrder request under the asserted identity. The identity is a RACF id contained in a UsernameToken which is transported as a SOAP header in the request message.

In this section, we document the following steps:

- ▶ Enabling CICS to use WS-Security support
- ▶ Defining a new pipeline to process placeOrder request messages
- ▶ Authorizing the service requester to run the placeOrder service

8.4.1 Enabling CICS to use WS-Security support

To implement the full range of CICS WS-Security processing, you must apply a number of updates to your CICS region. We document these steps in “Enabling CICS for WS-Security processing” on page 47.

Note: We discovered that it was not necessary to add the hlq.SDFHWSLD library to the DFHRPL concatenation for this security scenario.

8.4.2 Defining a new pipeline

Since the inquireCatalog and inquireSingle service requests will not be subject to identity assertion processing, we need to create a new pipeline specifically for the placeOrder requests.

Creating the pipeline configuration file

Example 8-3 shows the pipeline configuration file that we used for the identity assertion test scenario. It includes the user-written handler CIWSMSGO and configuration information for the CICS-supplied security handler.

Example 8-3 Pipeline config file, ITSO_7658_wssec_idassertionuser_ID_provider.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
  provider.xsd ">
  <service>
    <service_handler_list>
      <handler>
        <program>SNIFFER</program>
        <handler_parameter_list/>
      </handler>
      <wsse_handler>
        <dfhwsse_configuration version="1">
          <authentication trust="blind" mode="basic">
            </authentication>
          </dfhwsse_configuration>
        </wsse_handler>
      <handler>
        <program>CIWSMSGO</program>
        <handler_parameter_list/>
      </handler>
    </service_handler_list>
  </terminal_handler>
</provider_pipeline>
```

```
        <cics_soap_1.2_handler/>
    </terminal_handler>
</service>
<apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

The `<wsse_handler>` element shown in Example 8-3 on page 273 contains a `<dfhwsse_configuration>` element that specifies configuration information for the CICS-supplied security handler.

In the `<authentication>` element

- ▶ The `trust="blind"` attribute specifies that asserted identity is used.
- ▶ The `mode="basic"` attribute specifies that inbound messages must contain an identity token, where the identity token contains a user ID and optionally a password.

CICS puts the user ID in the DFHWS-USERID container. If no password is included, CICS uses the user ID without verifying it. If a password is included, the CICS-supplied security handler verifies it.

The CIWSMSGO message handler program changes the transaction ID to ORDS, the secure order transaction.

Note: Another potential use of a custom message handler is to write an audit record each time a service is requested, for example, a message could be written to a CICS user journal.

Creating a new pipeline to process placeOrder requests

We now create the new pipeline resource definition EXSECURE and we associate this pipeline with the configuration file `ITSO_7658_wssec_Idassertionuser ID_provider.xml` and the `wsbind` directory `/CIWS/S3C3/wsbind/provider/secured`.

We do this using the RDO commands in Example 8-4 and then re-installing the PIPELINE resource definition.

Example 8-4 Create pipeline for signature processing

```
CEDA COPY PIPELINE(EXPIPE01) GROUP(S3C2) AS(EXSECURE)
CEDA ALTER PIPELINE(EXSECURE) GROUP(S3C3) Configfile(/CIWS/S3C3/config/
ITSO_7658_wssec_Idassertionuser ID_provider.xml)
Wsdir(/CIWS/S3C3/wsbind/provider/secured/)
```

We move the placeOrder.wsbind file to the /CIWS/S3C3/wsbind/provider/secured/ directory so that the placeOrder Web service is dynamically installed when the EXSECURE pipeline is installed.

8.4.3 Authorizing the service requester

In this section we show the RACF commands that are used to authorize the service requester to run the placeOrder Web service.

Permitting access

In “Setting up basic security configuration” on page 161 we show the RACF commands that we used to permit the pre-defined user ID (PRIVAT01) to start the ORDR transaction.

Example 8-5 shows the RACF commands that we used to permit the service requester’s user ID (USERWS03) to start the ORDS transaction.

Example 8-5 RACF command to allow USERWS03 to run ORDS transaction

```
PERMIT CIWS3D.ORDS CLASS(TCICSTRN) ID(USERWS03) ACCESS(READ)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

Permitting surrogate access

Example 8-6 shows the RACF commands that we used to permit the pre-defined user ID (PRIVAT01) to start transactions (such as ORDS) with the user ID USERWS03.

Example 8-6 RACF commands to allow PRIVAT01 to act as surrogate for USERWS03

```
RDEFINE SURROGAT USERWS03.DFHSTART UACC(NONE) OWNER(CICSR6)
PERMIT USERWS03.DFHSTART CLASS(SURROGAT) ID(PRIVAT01) ACCESS(READ)
SETROPTS RACLIST(SURROGAT) REFRESH
```

For more information regarding surrogate user security, see *CICS Transaction Server for z/OS RACF Security Guide*, SC34-6835.

8.5 Testing the identity assertion scenario

In this section we show the results of testing the scenario. We show the following:

- ▶ Running the Web application

- ▶ Request from WebSphere to CICS captured with TCP/IP Monitor
- ▶ CEMT showing that the tasks run under the correct user IDs
- ▶ Authorization errors

Running the Web application

To configure the Web client application we enter the following URL:

http://mve2.pssc.ibm.com:8148/ExampleAppClientV6Web

The Web browser basic authentication pop-up window is displayed (Figure 8-6). We specify user ID **USERWS03** and a valid password. We note that if we specified an invalid password the basic authentication pop-up was re-displayed.

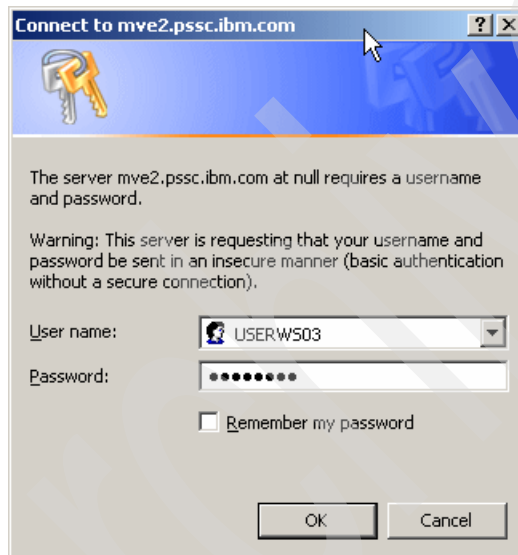


Figure 8-6 Catalog application basic authentication login

We click **OK**, and the Welcome page is displayed. We click the **CONFIGURE** so in order to set the service endpoints.

We configure the endpoints with hostname **wtsc66.itso.ibm.com** and port **14303** (Figure 8-7).

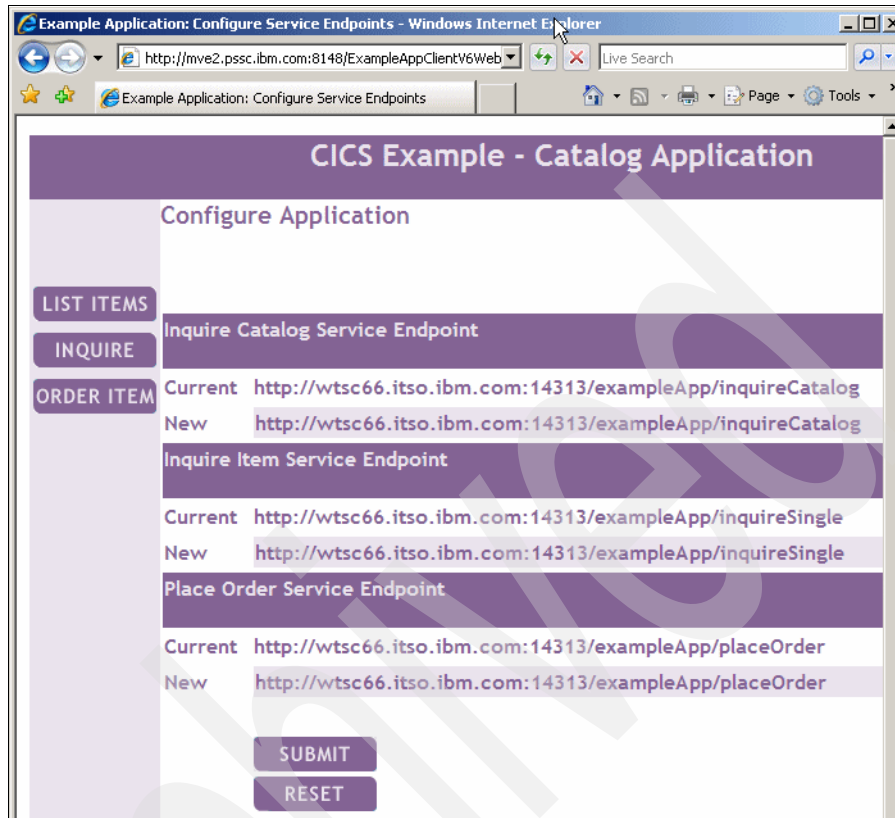


Figure 8-7 Configuring the CICS catalog manager example application

We click **SUBMIT** to complete the configuration.

We then click **ORDER** and the Enter Order Details page is displayed. We submit an order and verify that the order is successfully placed.

Request from WebSphere to CICS

Example 8-7 shows the security header part of the SOAP request message passed from WebSphere Application Server for z/OS and CICS.

Example 8-7 SOAP request message with asserted user ID

```
<soapenv:Header>
  <wsse:Security soapenv:mustUnderstand="1"
    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
    curity-secext-1.0.xsd">
    <wsse:UsernameToken>
      <wsse:Username>USERWS03</wsse:Username>
    </wsse:UsernameToken>
  </wsse:Security>
</soapenv:Header>
```

```
</wsse:UsernameToken>
</wsse:Security>
</soapenv:Header>
```

- ▶ The header includes the `mustUnderstand="1"` attribute, which indicates that either this header must be processed or a SOAP fault thrown.
- ▶ The `<wsse:UsernameToken>` contains the security credentials that we pass into CICS. There is one child element.
 - The `<wsse:Username>` element contains the asserted user ID `USERWS03`.

Note: There is no `<wsse:Password>` element in the `UsernameToken`.

CEMT INQUIRE TASK

Figure 8-8 shows the ORDS transaction running with user ID `USERWS03` (the asserted user ID), while the `ORDR` transaction runs with the `PRIVAT01` user ID.

```
INQUIRE TASK
STATUS: RESULTS - OVERTYPE TO MODIFY
Tas(0000057) Tra(CEMT) Fac(E041) Run Ter Pri( 255 )
  Sta(TO) Use(CICRSR6 ) Uow(C2BAB4C6A6D5FCC9)
Tas(0000061) Tra(ORDR)          Sus Tas Pri( 001 )
  Sta(U ) Use(PRIVAT01) Uow(C2BAB61C2ECD1FD4) Hty(RZCBNOTI)
Tas(0000062) Tra(ORDS)          Sus Tas Pri( 001 )
  Sta(U ) Use(USERWS03) Uow(C2BAB61C2FA23D0E) Hty(EDF      )
Tas(0000064) Tra(CEDF) Fac(E005) Sus Ter Pri( 001 )
  Sta(SD) Use(CICRSR6 ) Uow(C2BAB61C3707BCCC) Hty(ZCIIOWAIT)
                                     SYSID=S3C3 APPLID=A6POS3C3
```

Figure 8-8 ORDS executing with asserted user ID

Authorization errors

Example 8-8 shows the WebSphere and RACF error messages that are written to the system log if the user is not authorized to run the Web application.

Example 8-8 Unauthorized access to Catalog Web application

```
ICH408I USER(USERWS03) GROUP(SYS1) NAME(USERWS03)
W1.CATALOG CL(EJBROLE )
INSUFFICIENT ACCESS AUTHORITY
ACCESS INTENT(READ ) ACCESS ALLOWED(NONE )
+BB000222I: SECJ0129E: Authorization failed for USERWS03 while
invoking GET on default_host:ExampleAppClientV6Web/Welcome.jsp,
Authorizationfailed, Not granted any of the required roles: CATALOG
```

Example 8-9 shows the RACF error message that is written to the system log if the user is not authorized to run the CICS service provider application.

Example 8-9 Unauthorized access to Catalog CICS application

```
ICH408I USER(USERWS03) GROUP(SYS1  ) NAME(USERWS03)
        CIWS3D.ORDS CL(TCICSTRN)
        INSUFFICIENT ACCESS AUTHORITY
        ACCESS INTENT(READ  ) ACCESS ALLOWED(NONE  )
```

Archived

Archived

Identity assertion with WebSphere DataPower

A WebSphere DataPower SOA appliance can be used in conjunction with CICS Web services to provide an additional layer of protection and to offload expensive operations such as an XML digital signature processing.

In this chapter, we show how DataPower can verify a signed SOAP message before forwarding the request message to a CICS service provider application. After DataPower verifies the XML digital signature, it then asserts the X509 certificate (that is contained in the original request message) to CICS in an unsigned message.

9.1 Scenario overview

Figure 9-1 shows an overview of the WebSphere DataPower security scenario.

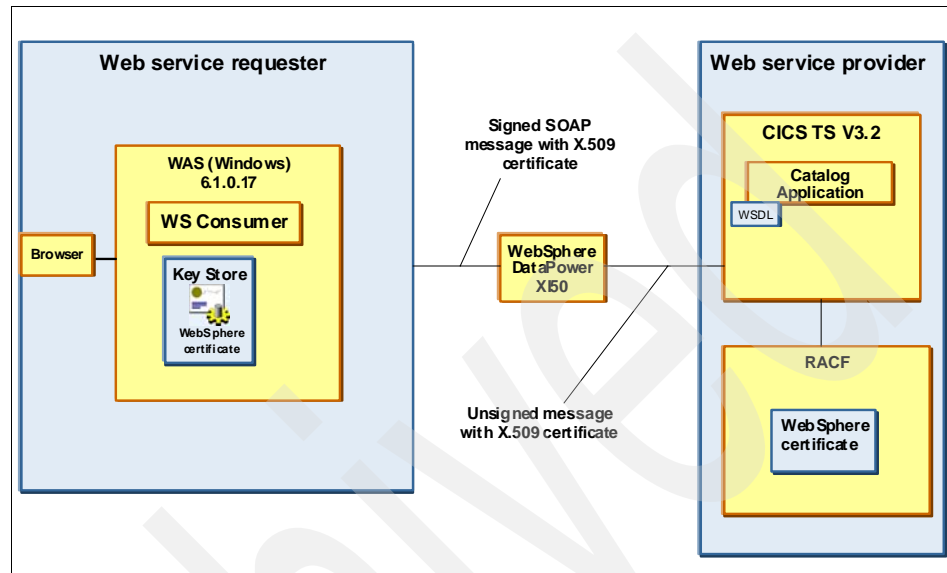


Figure 9-1 WebSphere DataPower scenario

The sequence of steps is as follows:

1. A user accesses WebSphere Application Server through a Web browser to place an order.
2. The service requester (WebSphere in our example) generates a BinarySecurityToken element from the WebSphere X.509 certificate. It then signs the message with its private key, and sends the request to the target endpoint which is configured to be the DataPower appliance.
3. DataPower verifies the XML digital signature, strips the signature from the message and forwards the X.509 certificate (that is contained in the original request message) to CICS in an unsigned message.
4. CICS receives the SOAP request through its TCPIP SERVICE and looks up the matching URIMAP. The URIMAP instructs CICS that requests for the path /exampleApp/placeOrder are to use the WEBSERVICE placeOrder, the PIPELINE EXSECURE, the transaction ORDR and the user ID PRIVAT01. CICS starts pipeline processing using this information.

The context containers DFHWS-USERID and DFHWS-TRANID are initialized with the current user ID, PRIVAT01, and the current tranid, ORDR.

5. The EXSECURE pipeline includes the CICS-supplied security handler, which is configured for identity assertion with blind trust and an authentication mode of signature. CICS calls RACF to map the WebSphere X.509 certificate to a RACF user ID, and puts the mapped user ID USERWS01 in the DFHWS-USERID container.

Note: In an identity assertion scenario like this one, it is normally necessary to establish a trust relationship, for example, by using SSL client authentication. We do not include configuration information for establishing SSL connectivity between DataPower and CICS in this chapter, but CICS SSL configuration is covered in detail in “Enabling SSL” on page 181.

The pipeline also includes the user-written message handler CIWSMSGO which sets the context container DFHWS-TRANID to contain ORDS for any placeOrder request.

6. The ORDS transaction runs under the asserted user ID USERWS01, and all further resource security checks for this request (for example, program and database security) are then performed against the USERWS01 user ID.

9.2 Preparation

The software we used is listed in Table 9-1.

Table 9-1 Software and firmware used in the DataPower scenario

Windows	DataPower	z/OS
Internet Explorer V7.0	Firmware version XI50.3.6.1.5 / Build:156262	z/OS V1.9
Windows XP Professional Version 2002 SP3		CICS Transaction Server V3.2
IBM WebSphere Application Server V6.1.0.17		
IBM WebSphere Application Server Toolkit V6.1.1.6		

Windows	DataPower	z/OS
Our J2EE applications <ul style="list-style-type: none"> ▶ ExampleAppClientV6.ear The Web client for the CICS catalog example application supplied with CICS TS. 	Our XSLT script <ul style="list-style-type: none"> ▶ ExtractCertificateFromSignature.xml The XSLT used to remove the signature from the SOAP request. 	Our user-written CICS message handler programs <ul style="list-style-type: none"> ▶ CIWSMSGO This program changes the transaction ID for placeOrder requests to ORDS. ▶ SNIFFER (message handler program) This program browses through the containers available in the pipeline.

9.2.1 Definition checklist

The definitions we used are listed in Table 9-2.

Table 9-2 Settings used in DataPower scenarios

Value	CICS TS	WebSphere Application Server	DataPower
IP name	wtsc66.itso.ibm.com	Cev8-Pc3.pssc.mop.fr.ibm.com	
IP address	9.12.4.75		9.212.133.4
TCP/IP port	14314	9080	9080
Jobname	CIWSS3C4		
APPLID	A6POS3C4		
TCPIP SERVICE	S3C4		
Provider PIPELINES	EXPIPE01 for inquireSingle and inquireCatalog services EXSECURE for placeOrder service		

Value	CICS TS	WebSphere Application Server	DataPower
Configuration files	ITSO_7658_basicsoap12 provider.xml for inquireSingle and inquireCatalog services ITSO_7658_wssec_IdassertionCertificate_provider.xml for placeOrder service		

9.2.2 User IDs

The user IDs we used in our configuration are listed in Table 9-3.

Table 9-3 User IDs used in DataPower scenario

Value	CICS TS
CICS region user ID	CIWS3D
User ID for which we wish to permit access to inquireSingle and inquireCatalog services	PUBLIC01
User ID for which we wish to permit access to the ORDR transaction of the placeOrder service	PRIVAT01
User ID for which we wish to permit access to the ORDS transaction of the placeOrder service	USERWS01

9.2.3 Certificates

The certificates we used in our configuration are listed in Table 9-4.

Table 9-4 Certificates used in DataPower scenario

Value	Format	File
WebSphere CA certificate	CERTDER	WASWINROOT.CER
WebSphere server certificate	PKCS12DER	WWSERV01.P12

9.3 Configuration

We document the following procedures to enable the setup of the DataPower security scenario:

- ▶ Configuring the WebSphere service requester
- ▶ Configuring DataPower
- ▶ Configuring CICS

9.3.1 Configuring the WebSphere service requester

The WebSphere configuration for this scenario is the same as that used in the XML digital signature scenario documented in Chapter 7, “Signing the SOAP message” on page 221.

9.3.2 Configuring DataPower

In this section we show how to configure DataPower, in particular:

- ▶ How to define the Web service proxy for the placeOrder Web service
- ▶ How to verify the signed SOAP message
- ▶ How to strip the signature part of the message and forward the unsigned message to CICS

We document these procedures:

- ▶ Accessing the DataPower control panel
- ▶ Creating DataPower objects
- ▶ Configuring an HTTP front side handler
- ▶ Adding certificates to DataPower
- ▶ Configuring a Web service proxy
- ▶ Verifying the XML digital signature
- ▶ Stripping the XML digital signature

Accessing the DataPower control panel

We use the following URL to access the DataPower control panel:

<https://9.212.130.128:9090/login.xml>

Figure 9-2 shows the control panel.

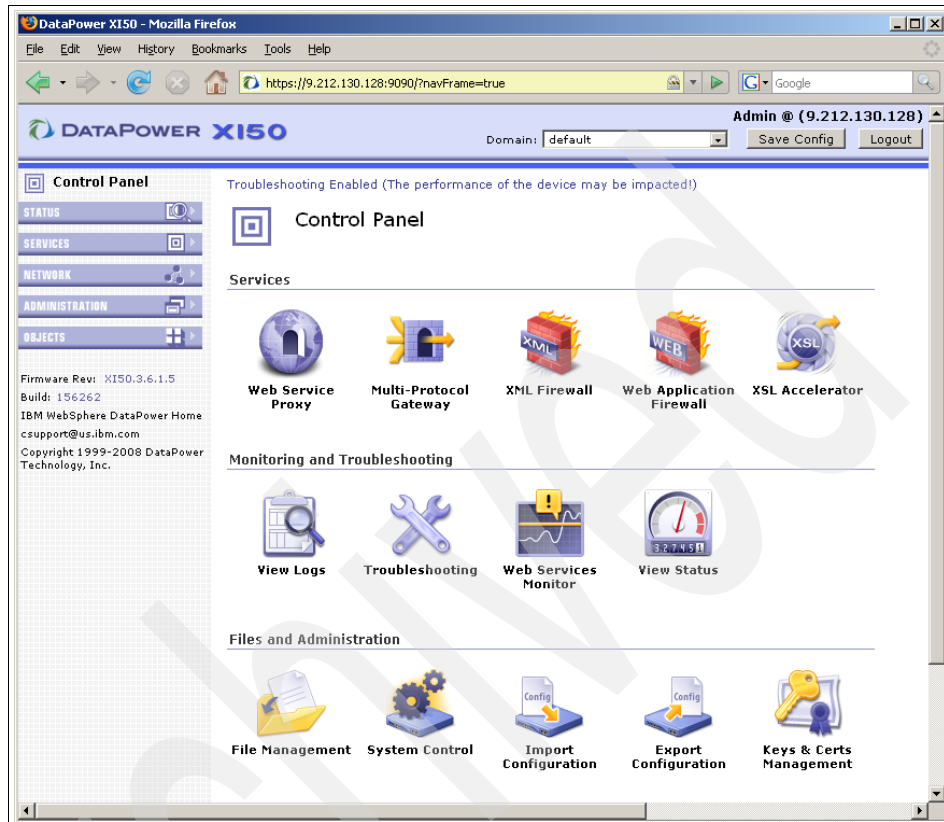


Figure 9-2 DataPower Control panel

Our DataPower appliance is configured with two network adapters:

- ▶ eth4 for management traffic
- ▶ eth0 for normal traffic.

Figure 9-3 shows the configured ethernet interface.

Control Panel

Troubleshooting Enabled (The performance of the device may be impacted!)

Configure Ethernet Interface

[Refresh List](#)

Name	Status	Op-State	Logs	IP Address	MTU	Use DHCP
eth0	modified	up		9.212.133.4/25	1400	off
eth1	saved	up			1500	off
eth2	saved	up			1500	off
eth4	modified	up		9.212.130.128/22	1500	off

Figure 9-3 DataPower ethernet interface

Note: It is a good practice from a security perspective to separate the management traffic from the general traffic.

DataPower objects

DataPower configuration items are internally represented as objects for reuse. Figure 9-4 outlines the objects that we configured for our scenario.

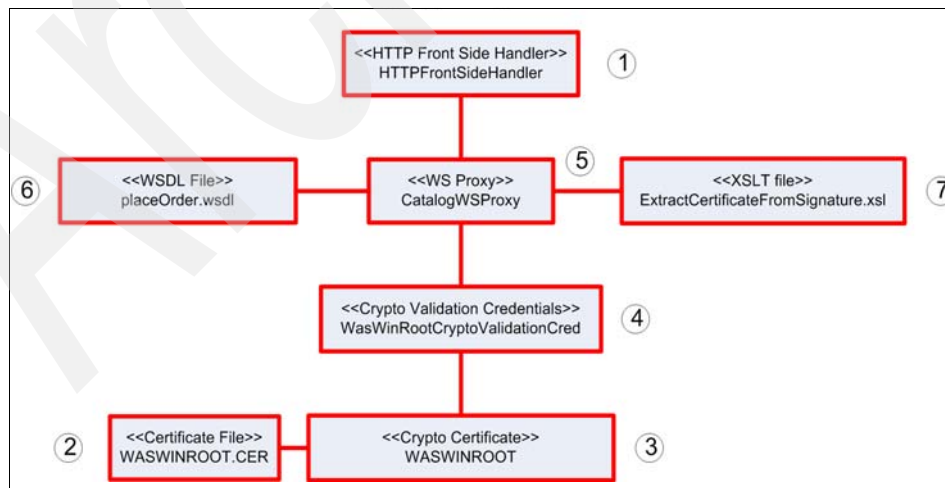


Figure 9-4 DataPower objects

An HTTP Front Side Handler (1) object handles HTTP protocol communications with HTTP clients. We create one to handle all HTTP requests. It listens on port 9080.

We use the WAXWINROOT.CER (2) file that contains the root certificate associated to the signer certificate, and we create a Crypto Certificate (3).

A Crypto Validation Credential (4) includes a collection of certificate objects used for authenticating presented credentials and verifying XML digital signatures.

We create a WS Proxy (5) using the placeOrder.wsdl (6) that intercepts Web service calls to the CICS placeOrder service.

We use a custom XSLT (7) to strip the signature from the request message.

Note: XSLT is an XML-based language used for the transformation of XML documents into other XML or “human-readable” documents. The original document is not changed; rather, a new document is created based on the content of an existing one.

Configuring an HTTP front side handler

An HTTP Front Side Handler object handles HTTP protocol communications. We add a local Endpoint Handler using the following steps:

1. On the DataPower control panel, expand **Objects**, and then click **HTTP Front Side Handler** under the **Protocol Handlers** section.

The list of HTTP Front Side Handlers is displayed. It should be empty at this stage as shown in Figure 9-5.

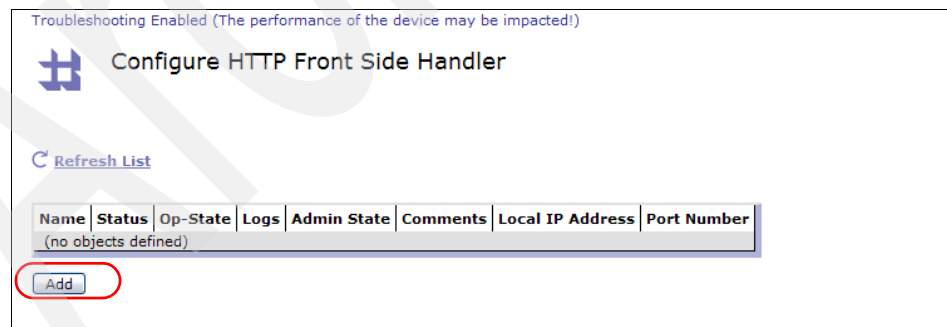


Figure 9-5 DataPower - configure HTTP Front Side Handler

2. Click **Add**.

3. On the **HTTP Front Side Handler** page, specify the properties of the Local Endpoint Handler:
 - Enter HTTPFrontSideHandler in the name field.
 - Enter the port number 9080 in the Port Number field.
 - Leave the other properties to default.

Figure 9-6 shows our configured HTTP Front Side Handler.

HTTP Front Side Handler

Apply Cancel Help

Name HTTPFrontSideHandler *

Admin State enabled disabled

Comments

Local IP Address 0.0.0.0 Select Alias *

Port Number 9080 *

HTTP Version to Client HTTP 1.1

Allowed Methods and Versions

- HTTP 1.0
- HTTP 1.1
- POST method
- GET method
- PUT method
- HEAD method
- OPTIONS
- TRACE method
- DELETE method
- URL with Query Strings
- URL with Fragment Identifiers
- URL with ..
- URL with cmd.exe

Persistent Connections on off

Compression on off

Maximum Allowed URL Length 16384

Maximum Allowed Total Header Length 128000

Maximum Number of HTTP Request Headers Allowed 0

Maximum Allowed Length of HTTP Header Name 0

Maximum Allowed Length of HTTP Header Value 0

Maximum Allowed Length of HTTP Query String 0

Access Control List (none) + ...

Figure 9-6 DataPower HTTP Front Side Handler

4. Click **Apply**.

Adding certificates to DataPower

Next we create a Crypto Certificate by importing the WebSphere CA certificate to DataPower and we create the Crypto Validation Credentials.

Configuring a Crypto Certificate

A Crypto Certificate defines a certificate object that specifies a public key and key alias. When DataPower receives a signed SOAP request, it validates the signature and also the validity of the certificate, particularly the trust associated to this certificate.

Using the following steps, we load the WebSphere CA certificate that we created in “Creating the WebSphere CA Certificate” on page 163.

1. On the DataPower control panel, expand **Objects**, and then click **Crypto Certificate** under the **Crypto** section.

The list of Crypto Certificates is displayed. It should be empty at this stage, as shown in Figure 9-7.

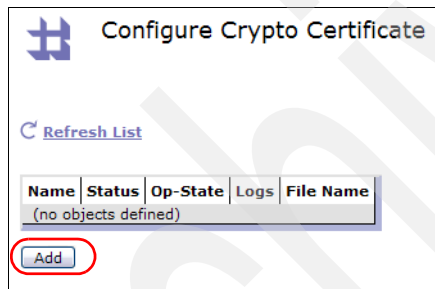


Figure 9-7 DataPower Crypto Certificates

2. Click **Add**.

3. Specify the file location of the WebSphere CA certificate (WASWINROOT.CER) and specify **Upload** (Figure 9-8).

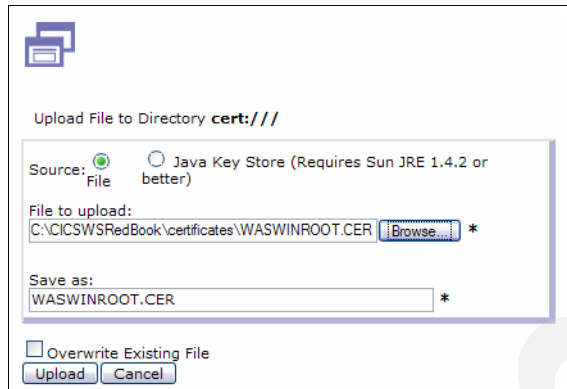


Figure 9-8 DataPower - adding Crypto certificate

4. A confirmation message that the certificate is successfully loaded is shown in Figure 9-9.



Figure 9-9 DataPower - confirmation of certificate upload

5. Click **Continue**.
6. Ensure that the WASWINROOT.CER file is selected for the file name, and enter the password to open the CER file (Figure 9-10).

Configure Crypto Certificate

This configuration has been modified, but not yet saved.

Main

Crypto Certificate

[Apply](#) [Cancel](#) [Help](#)

Name: WASWINROOT *

Admin State: enabled disabled

File Name: cert:/// WASWINROOT.CER [Details...](#) [Upload...](#) [Fetch...](#) *

Password:

Confirm Password:

Password Alias: on off

Ignore Expiration Dates: on off

Figure 9-10 DataPower - saving the certificate

7. Click **Apply** and the new Crypto Certificate is displayed (Figure 9-11).

Configure Crypto Certificate

[Refresh List](#)

Name	Status	Op-State	Logs	File Name
WASWINROOT	new	up		cert:///WASWINROOT.CER

[Add](#)

Figure 9-11 DataPower - configured Crypto certificate

Configuring the Crypto Validation Credentials

A Crypto Validation Credential defines a validation credentials list object that includes a collection of certificate objects used for authenticating presented credentials and validating XML digital signatures.

We create the Crypto Validation Credentials using the following steps:

1. On the DataPower control panel, expand **Objects**, and then click **Crypto Validation Credentials** under the **Crypto** section.

The list of Crypto Validation Credentials is displayed. It should be empty at this stage as shown in Figure 9-12.

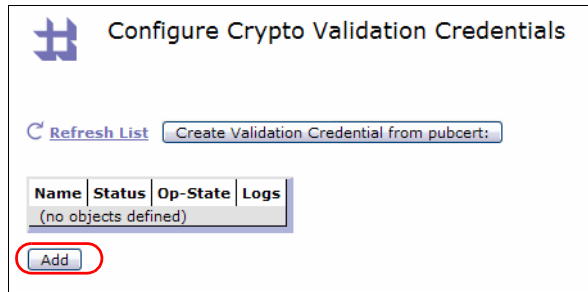


Figure 9-12 DataPower Crypto Validation Credentials

2. Click **Add**.
3. On the **Configure Crypto Validation Credentials** page, specify the properties of the credentials:
 - Enter WasWinRootCryptoValidationCred in the name field.
 - Select WASWINROOT in the Certificates drop-down list.
 - Leave the other properties to default.

Figure 9-13 shows our configured Crypto Validation Credential.

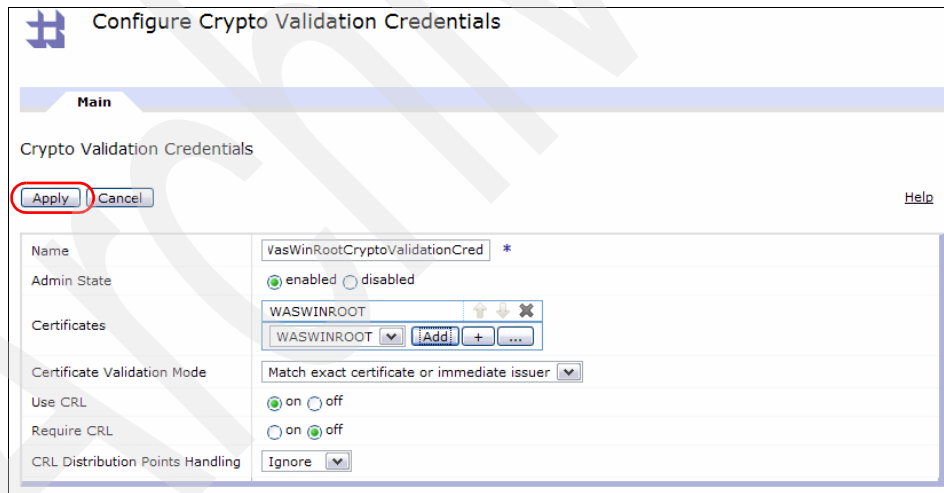


Figure 9-13 DataPower - saving the Crypto Validation Credentials

4. Click **Apply** and the new Crypto Validation Credential is displayed (Figure 9-14).

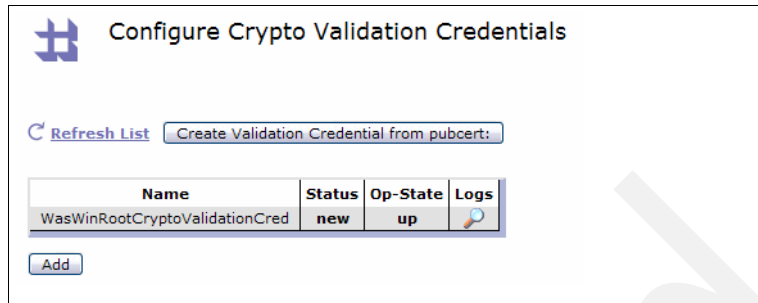


Figure 9-14 DataPower - configured Crypto Validation Credential

Configuring a Web Service Proxy

In this section, we create a simple Web Service Proxy Service that intercepts Web service calls to the CICS placeOrder service. A Web service proxy provides:

- ▶ A facade for arbitrary back-end services.
- ▶ Quick service virtualization by simply uploading WSDL into the DataPower device.
- ▶ A “living” virtual service that passes messages between the client and the real service, so that the client connects to the proxy and not to the back-end service.

In doing this, the service consumer and the service provider do not need to be tightly coupled and bound to each other. The DataPower appliance can hide the details of accessing the service from the consumer.

We create the Web Service Proxy using the following steps:

1. On the DataPower control panel, expand **Services**, and then click **New Web Service Proxy**, under the **Web Service Proxy** section.
2. On the Configure Web Service Proxy panel, enter **CatalogWSProxy** for the name of the Web Service Proxy (Figure 9-15).

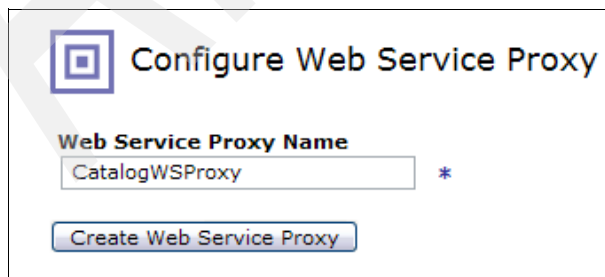


Figure 9-15 DataPower - configure Web Service Proxy

3. Click **Create Web Service Proxy**.
4. To create the WS Proxy, we need to upload one WSDL file associated to this WS Proxy. We are going to load the placeOrder.wsdl file that is located on our local file system.
5. Specify the file location of the placeOrder.wsdl file and specify **Upload** (Figure 9-16).

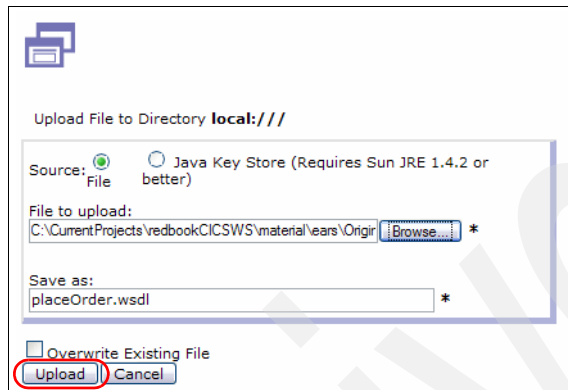


Figure 9-16 DataPower - uploading WSDL file

6. A confirmation message that the WSDL file is successfully loaded is shown in Figure 9-17.

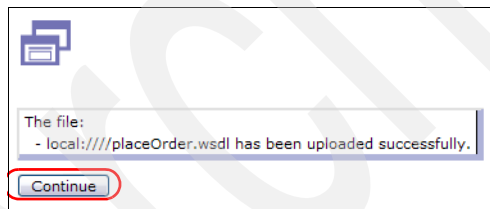


Figure 9-17 DataPower - uploaded WSDL file

7. Click **Continue**. Figure 9-18 shows the configured Web Service Proxy.

Configure Web Service Proxy

WSDLs | SLM | Services | Policy | Proxy Settings | Advanced Proxy Settings | Headers/Params

Web Service Proxy Name [up]
CatalogWSProxy *

Apply | Cancel | Delete | Refresh | [View Log](#) | [View Status](#) | [View Operations](#) | [Show Probe](#) | [Validate Conformance](#) | [Help](#)

Web Service Proxy WSDLs

Edit WSDL/Subscription
 Add WSDL
 Add UDDI Subscription
 Add WSRR Subscription

WSDL File URL
 local:/// placeOrder.wsdl | Upload... | Fetch... | Browse UDDI *

Use WS-Policy References
 on off *

WS-Policy Parameter Set
 (none) | + | ...

WS-Policy Enforcement Mode
 enforce | v

Next

Figure 9-18 DataPower - configured Web Service Proxy

8. Click **Next**.
9. On the next panel (Figure 9-19) we associate the Web Service Proxy with the HTTP Front Side Handler that we configured in “Configuring an HTTP front side handler” on page 289 and we specify the endpoint information for the target CICS placeOrder service.

Configure Web Service Proxy [Help](#)

WSDLs | SLM | Services | Policy | Proxy Settings | Advanced Proxy Settings | Headers/Params

Web Service Proxy Name [up]
CatalogWSProxy *

Web Service Proxy WSDLs

DFH0XCMNService3 - DFH0XCMNPort

Local

Local Endpoint Handler	URI	Binding (Suffix)	Edit/Remove
HTTPFrontSideHandler	/exampleApp/placeOrder	soap-11()	Edit Remove
HTTPFrontSideHandl	/exampleApp/placeOrder	<input checked="" type="checkbox"/> SOAP 1.1 <input type="checkbox"/> SOAP 1.2 <input type="checkbox"/> HTTP GET	+ Add

Remote

Protocol	Hostname (IP Address)	Port	Remote URI
http	9.12.4.75	14303	/exampleApp/placeOrder

Published Use Local

Figure 9-19 DataPower - add front side handler and target service endpoint information

Select the HTTPFrontSideHandler and click the **+Add** link, so it appears in the list of Local Endpoint Handlers.

Specify the endpoint information for the target CICS service:

- Select `http` as the protocol.
- Enter `9.12.4.75` in the hostname (IP Address) field.
- Enter `14303` in the Port field.
- Allow the Remote URI to default as `/exampleApp/placeOrder`

10. Click **Next** and then **Apply**.

Validation rules

DataPower allows you to specify policy rules for Web service proxies, for example, schema validation and publishing rules. By default, schema validation is enabled.

We found that the placeOrder request messages did not validate correctly against the provided schema. We therefore needed to turn schema validation off.

On the **Configure Web Service Proxy** page, we click the **Policy** tab. We then click the little box (red or green) next to `placeOrder.wsdl`. On the right, a window appears that allows the configuration of the validation rules. We uncheck **Schema validate request messages**, and then click **Done**. See Figure 9-20.

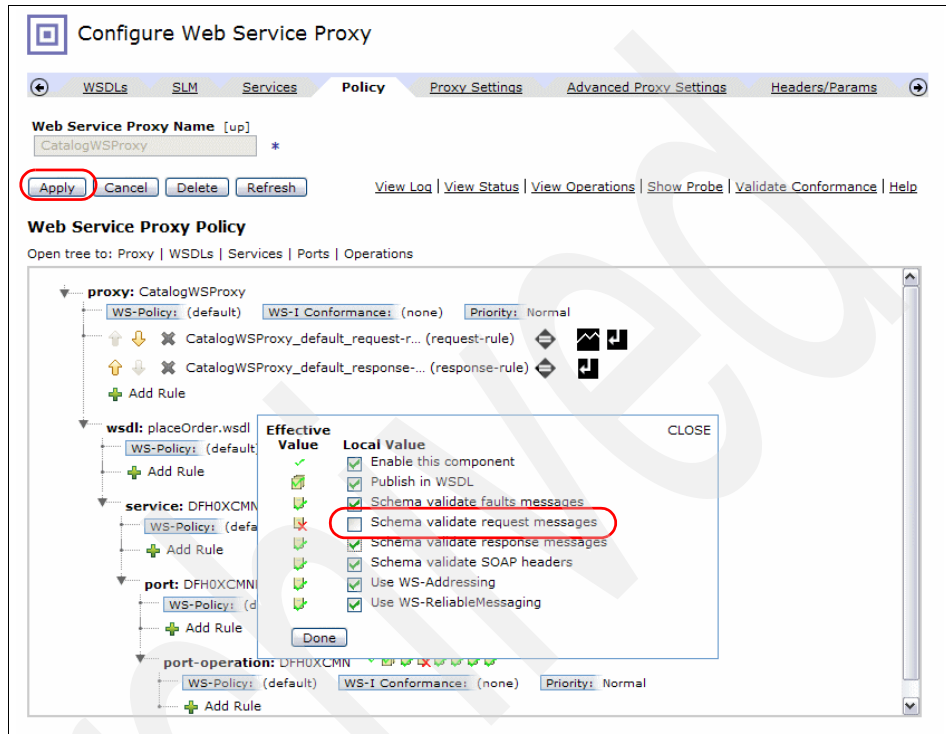


Figure 9-20 DataPower - disable schema validation

Click **Apply**. Figure 9-21 shows the completed Web service proxy configuration.

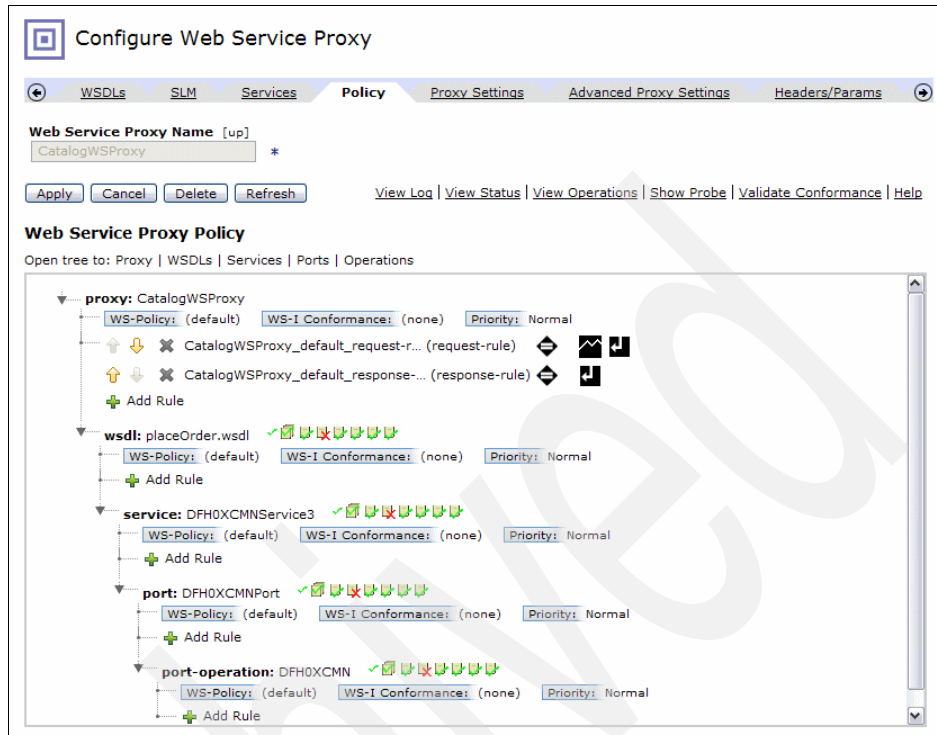


Figure 9-21 DataPower - disabled schema validation

Verifying the XML digital signature

Next we configure the signature verification. On the **Configure Web Service Proxy** page, we expand the `placeOrder.wsdl` to see the port-operation `DFH0XCMPN`, then click the **+ Add Rule**. The Rule Editor page opens.

We select the Rule Direction, in our case Client to Server. We drag and drop the Verify icon, on the right of the = signs, and double-click the icon. See Figure 9-22.

Note: In the rule editor, when the rule is not configured, there is a bold yellow border around the icon.

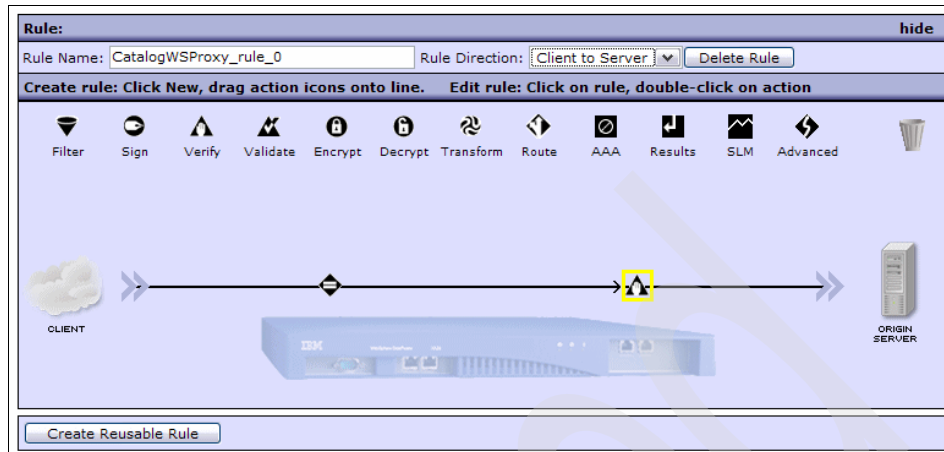


Figure 9-22 DataPower - signature verification

On the **Configure Verify Action** page, select the WasWinServerCryptoValidationCred Crypto from the Validation Credentials drop-down list, and click **Done**. See Figure 9-23.

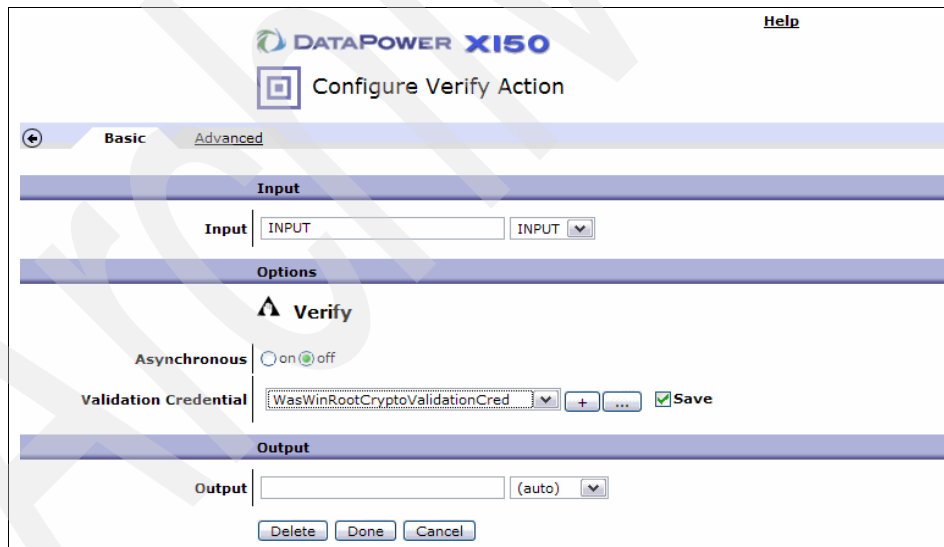


Figure 9-23 DataPower - configure verify action

Click **Apply** (on the top right of the panel) to apply the configuration.

Stripping the XML digital signature

A DataPower transformation can be used to transform an XML message to another XML message, or an XML message to a non-XML message. We use a transform rule to strip the XML digital signature from the request message.

To add the signature stripping transform rule, we drag and drop the **Transform** icon to the rule editor. See Figure 9-24.

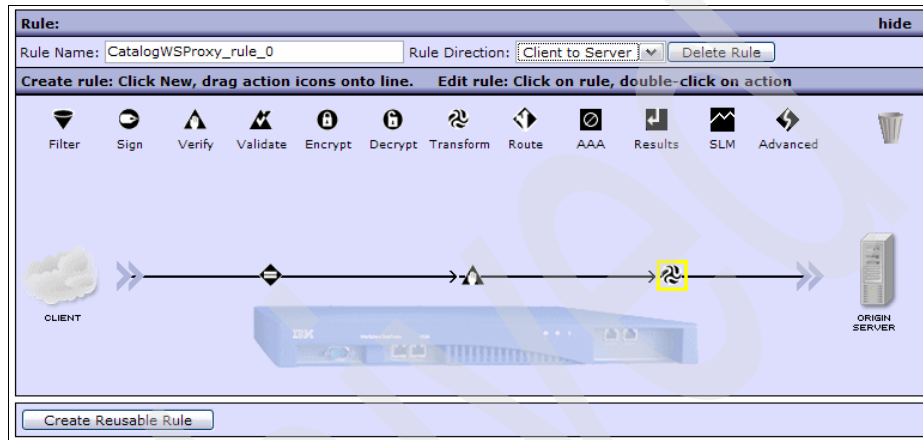


Figure 9-24 DataPower - add signature stripping transform rule

Browse on the local file system, find the XSLT file `xtractCertificateFromSignature.xsl` and then click **Upload**. See Figure 9-25.

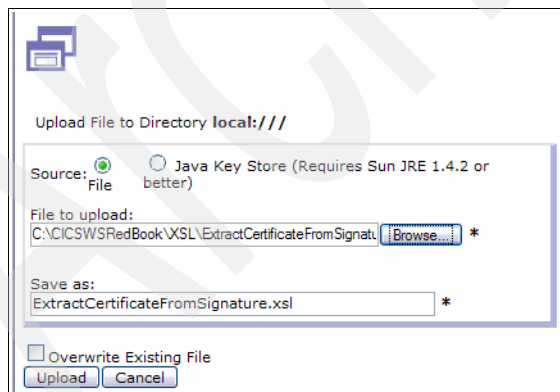


Figure 9-25 DataPower - XSLT file

Click **Apply** (on the top right of the panel) to apply the configuration. Click **SaveConfig** to save the configuration.

Note: The signature stripping XSLT file is shown in Appendix A, “XSLT example” on page 363.

9.3.3 Configuring CICS

In this section, we show how to configure the CICS pipeline to run the placeOrder request under the user ID associated with the WebSphere certificate contained in the request message.

In this section, we document the following steps:

1. Enabling CICS to use WS-Security support
2. Defining a new pipeline to process placeOrder request messages
3. Authorizing the service requester to run the placeOrder service

Enabling CICS to use WS-Security support

To implement Web Services Security, you must apply a number of updates to your CICS region.

Important: It is not necessary to have ICSF enabled in order to implement this scenario. A certificate that CICS uses to sign SOAP messages must be created with the ICSF option but CICS can receive certificates that are not created with an ICSF key.

Defining a new pipeline

Since the inquireCatalog and inquireSingle service requests will not be subject to XML signature processing, we need to create a new pipeline specifically for the placeOrder requests.

Creating the pipeline configuration file

Example 9-1 shows the pipeline configuration file that we used for the this test scenario. It includes the user-written handler CIWSMSGO and configuration information for the CICS-supplied security handler.

Example 9-1 Pipeline config file, ITSO_7658_wssec_ldassertionCertificate_provider.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
  provider.xsd ">
  <service>
```

```

<service_handler_list>
  <handler>
    <program>CIWSMSG0</program>
    <handler_parameter_list/>
  </handler>
  <handler>
    <program>SNIFFER</program>
    <handler_parameter_list/>
  </handler>
  <wsse_handler>
    <dfhwsse_configuration version="1">
      <authentication trust="blind" mode="signature">
      </authentication>
    </dfhwsse_configuration>
  </wsse_handler>
</service_handler_list>
</terminal_handler>
  <cics_soap_1.2_handler/>
</terminal_handler>
</service>
<apphandler>DFHPITP</apphandler>
</provider_pipeline>

```

The CIWSMSG0 message handler program changes the transaction ID to ORDS, the secure order transaction.

The `<wsse_handler>` element shown in Example 9-1 on page 303 contains a `<dfhwsse_configuration>` element that specifies configuration information for the CICS-supplied security handler.

In the `<authentication>` element:

- ▶ The `trust="blind"` attribute specifies that asserted identity is used.
- ▶ The `mode="signature"` attribute specifies that inbound messages must contain an X.509 certificate in a BinarySecurityToken.

If `<authentication mode="signature">` is specified, then the client X.509 certificate must be imported to RACF and attached to the CICS key ring because CICS runs the service request under the user ID associated with the client certificate.

The combination of the `trust="blind"` and `mode="signature"` attributes means that inbound messages must contain an identity token, where the identity token is the first X.509 certificate in the SOAP message header. The certificate does not need to have signed the message. The CICS-supplied security handler extracts the matching user ID and places it in the DFHWS-USERID container.

Creating a new pipeline to process placeOrder requests

We now create the new pipeline resource definition EXSECURE and we associate this pipeline with the configuration file ITSO_7568_wssec_IdassertionCertificate_provider.xml and the wsbind directory /CIWS/S3C2/wsbind/provider/secured.

We do this using the RDO commands in Example 9-2 and then installing the PIPELINE resource definition.

Example 9-2 Create pipeline for id assertion processing

```
CEDA COPY PIPELINE(EXPIPE01) GROUP(S3C4) AS(EXSECURE)
CEDA ALTER PIPELINE(EXSECURE) GROUP(S3C4) Configfile(/CIWS/S3C4/config/
ITSO_7568_wssec_IdassertionCertificate_provider.xml)
Wmdir(/CIWS/S3C4/wsbind/provider/secured/)
```

We move the placeOrder.wsbind file to the directory /CIWS/S3C4/wsbind/provider/secured/ so that the placeOrder Web service is dynamically installed when the EXSECURE pipeline is installed.

9.3.4 Authorizing the service requester

In this section we show the RACF commands that are used to authorize the service requester to run the placeOrder Web service.

Permitting access

In “Setting up basic security configuration” on page 161 we show the RACF commands that we used to permit the pre-defined user ID (PRIVAT01) to start the ORDR transaction.

Example 9-3 shows the RACF commands that we used to permit the caller’s user ID (USERWS01) to start the ORDS transaction.

Example 9-3 RACF command to allow USERWS01 to run ORDS transaction

```
PERMIT CIWS3D.ORDS CLASS(TCICSTRN) ID(USERWS01) ACCESS(READ)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

Permitting surrogate access

Example 9-4 shows the RACF commands that we used to permit the pre-defined user ID (PRIVAT01) to start transactions (such as ORDR) with the user ID USERWS01.

Example 9-4 RACF commands to allow PRIVAT01 to act as surrogate for USERWS01

```
RDEFINE SURROGAT USERWS01.DFHSTART UACC(NONE) OWNER(CICSR6)  
PERMIT USERWS01.DFHSTART CLASS(SURROGAT) ID(PRIVAT01) ACCESS(READ)  
SETROPTS RACLIST(SURROGAT) REFRESH
```

For more information regarding surrogate user security, see *CICS Transaction Server for z/OS RACF Security Guide*, SC34-6454.

9.4 Testing the Identity assertion with WebSphere DataPower scenario

In this section we show the results of testing the scenario. We show the following procedures:

- ▶ Running the Web application
- ▶ CEMT showing that the tasks run under the correct user IDs
- ▶ Using the Probe facility of DataPower

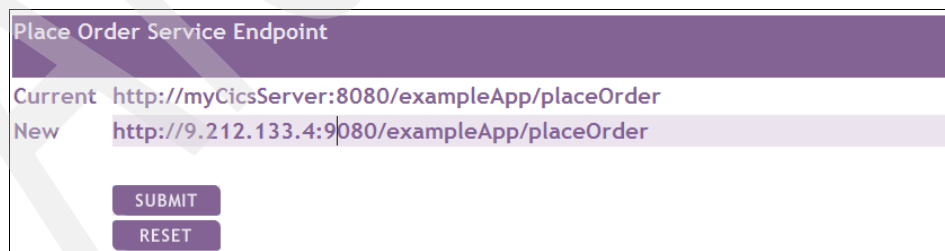
9.4.1 Running the Web Application

To configure the Web client application, we enter the following URL:

<http://Cev8-Pc3:9080/ExampleAppClientV6Web>

On the window presented in Figure 9-26, we enter the endpoint for the placeOrder service. The host address is set to **9.212.133.4** and the port is set to **9080**. These are the host address and port used by the DataPower appliance.

Click **Submit** to save the changes.



Place Order Service Endpoint	
Current	http://myCicsServer:8080/exampleApp/placeOrder
New	http://9.212.133.4:9080/exampleApp/placeOrder
<input type="button" value="SUBMIT"/>	
<input type="button" value="RESET"/>	

Figure 9-26 DataPower scenario - configure sample application

Click the **Order** button (not shown in figure) and enter any name for the User Name, and the Department Name, then click **Submit**. See Figure 9-27.

CICS Example - Catalog Application	
Enter Order Details	
LIST ITEMS	Item Reference Number
INQUIRE	0010
	Quantity
	001
	User Name
	John
	Department Name
	D034
	SUBMIT
BACK	
CONFIGURE	
CICS Transaction Server for z/OS	

Figure 9-27 DataPower scenario - testing the application

Figure 9-28 shows the successful response.

CICS Example - Catalog Application	
Order Placed	
LIST ITEMS	ORDER SUCESSFULLY PLACED
INQUIRE	
ORDER ITEM	

Figure 9-28 DataPower scenario - successful response.

CEMT INQUIRE TASK

Figure 9-29 shows the ORDS transaction running with the user ID USERWS01 (the user ID associated with the WebSphere certificate), while the ORDR transaction runs with the PRIVAT01 user ID. The WebSphere certificate is forwarded to CICS by DataPower in an unsigned request message.

```
INQUIRE TASK
STATUS: RESULTS - OVERTYPE TO MODIFY
Tas(0000075) Tra(ORDR)          Sus Tas Pri( 001 )
  Sta(U ) Use(PRIVAT01 ) Uow(BF3C2E9E23175284E) Hty(RZCBNOTI)
Tas(0000076) Tra(ORDS)          Sus Tas Pri( 001 )
  Sta(U ) Use(USERWS01 ) Uow(BF3C2E9E2497E51A) Hty(EDF )
Tas(0000078) Tra(CEDF) Fac(E024) Sus Ter Pri( 001 )
  Sta(SD) Use(CICRSR6 ) Uow(BF3C2E9E28D6A143) Hty(ZCIOWAIT)
Tas(0000079) Tra(CEMT) Fac(E025) Run Ter Pri( 255 )
  Sta(T0) Use(CICRSR6 ) Uow(BF3C2EA64298C1CB)

SYSID=S3C4 APPLID=A6POS3C4
```

Figure 9-29 ORDS executing with user ID associated with WebSphere certificate

9.4.2 Using the Probe facility of DataPower

The DataPower multi-step probe provides a very powerful feature to perform debugging and effective problem determination. We use the probe to view the message as it passes through the appliance.

The probe is enabled for a specific service within a domain. Click the **Troubleshooting** icon in the Control Panel. Then click the **Debug Probe** link as shown Figure 9-30.

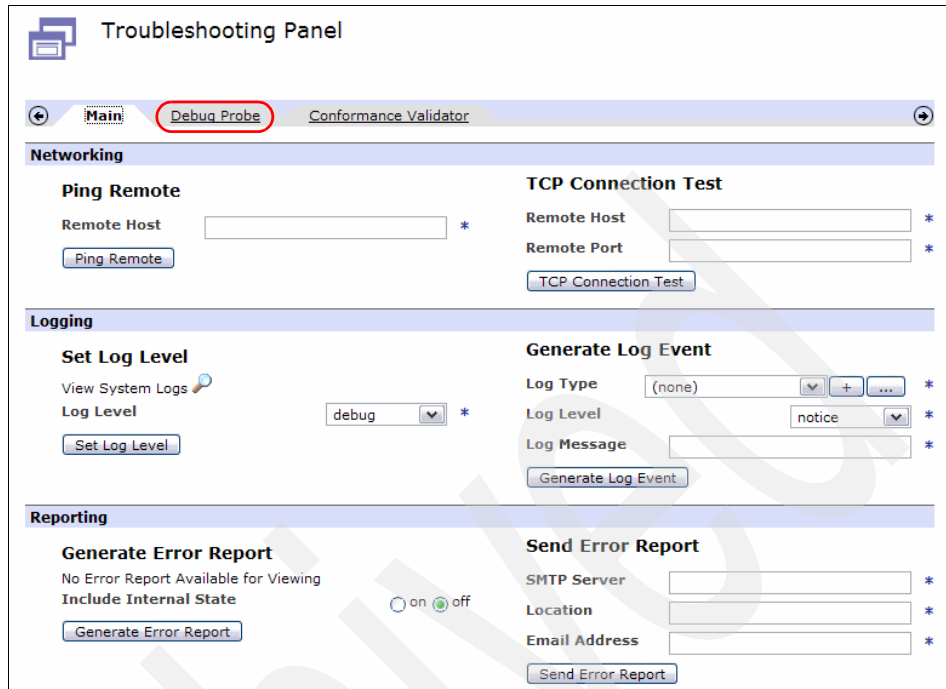


Figure 9-30 DataPower Troubleshooting Panel

Chose the CatalogWSProxy proxy service, and click **Add Probe** (Figure 9-31).



Figure 9-31 DataPower - enabling probe

After executing a request, the probe shows the request and response messages that have been traced by DataPower. We click the eyeglass of the request to view the details of the message (Figure 9-32).

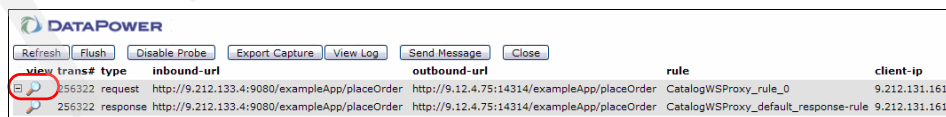


Figure 9-32 DataPower probe messages

Figure 9-33 shows the signed message received by DataPower. We see the BinarySecurityToken and signature parts of the Security SOAP header.

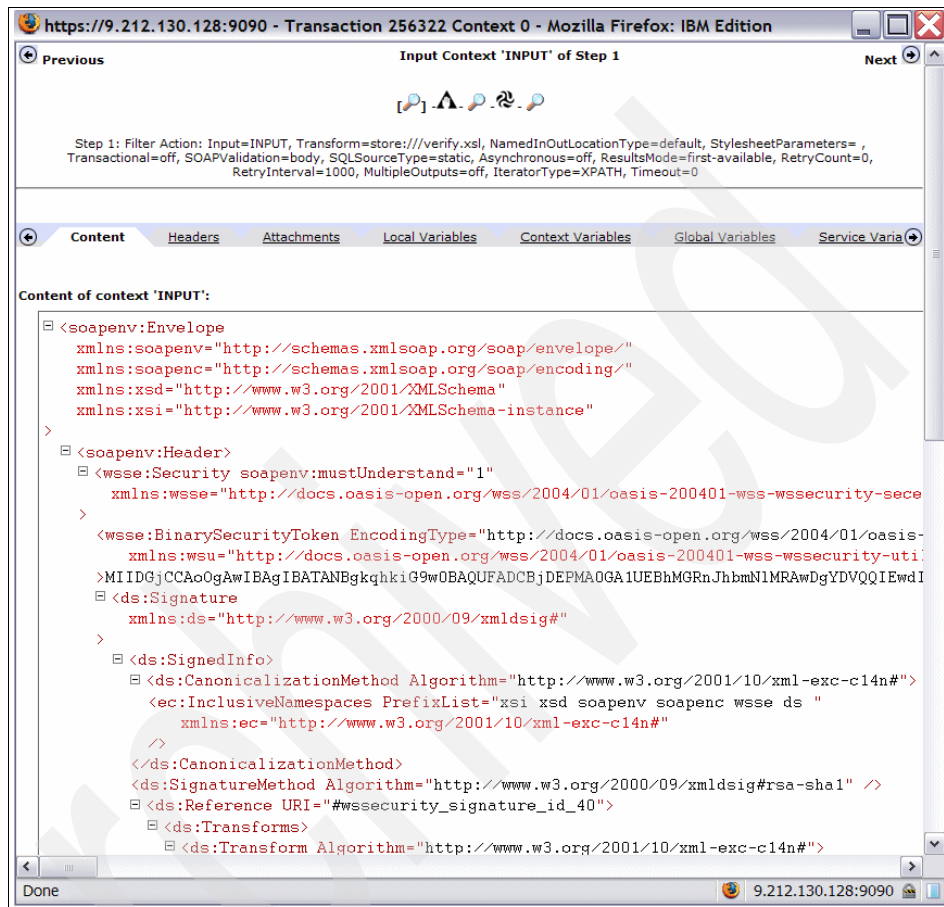


Figure 9-33 DataPower probe signed message

After the transform step, we can see that the BinarySecurityToken remains as part of the SOAP header, but that the signature part has been removed (Figure 9-34).

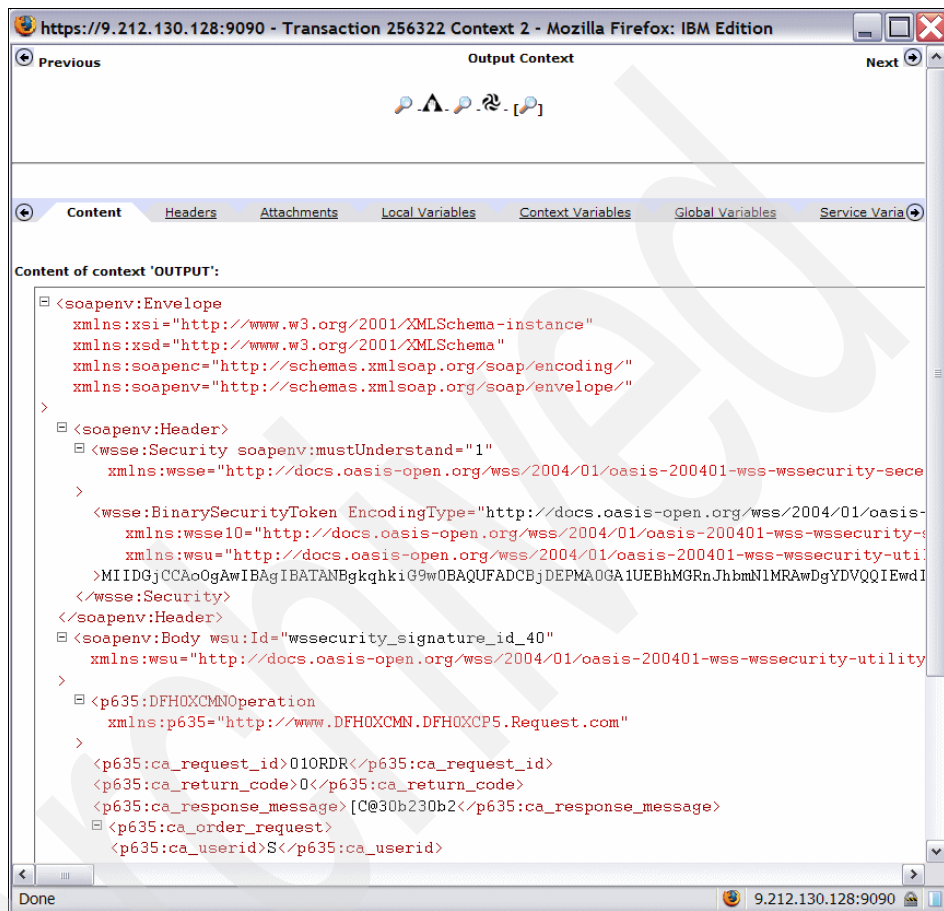


Figure 9-34 DataPower probe unsigned message

Enabling WS-Trust with TFIM

With CICS TS V3.2 you can verify or exchange security tokens with a Security Token Service (STS) using the WS-Trust (Web Services Trust Language) specification.

In this chapter we show how CICS implements support for the WS-Trust specification. We configure a pipeline in CICS to interoperate with an STS provided by Tivoli Federated Identity Manager (TFIM). This allows CICS to send and receive messages containing a variety of different security tokens, by using TFIM to validate identities that CICS does not itself recognize, and to exchange tokens of one type for another type.

10.1 Scenario overview

In our initial WS-Trust scenario, CICS receives a SOAP request message containing user credentials within a UsernameToken. The user ID associated with the UsernameToken is not known to CICS or RACF.

Note: In a second scenario (see “Using different token types” on page 350) we also show how CICS can process a SOAP request message that contains a token type that is not supported directly by CICS.

The placeOrder Web Service is secured by using CICS WS-Trust support in a secured provider pipeline.

We show how the WebSphere authenticated user (JOHNDOE) can be switched to a RACF user (USERWS05) known by CICS in order to run the secured transaction ORDS. This scenario is shown in Figure 10-1.

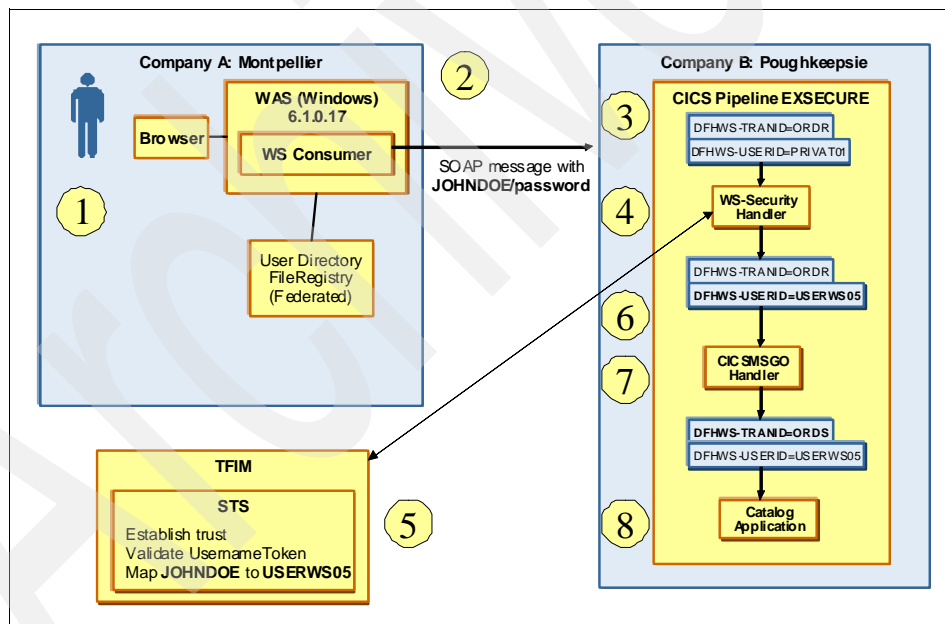


Figure 10-1 The WS-Trust scenario

The sequence of events is as follows:

1. A user accesses WebSphere Application Server through a Web browser to place an order.

2. WebSphere Application Server constructs a SOAP Request for the placeOrder Web Service. In the SOAP Headers it inserts a UsernameToken for Username JOHNDOE and Password password.

Note: In this scenario, the security credentials are configured in the EAR file.

3. CICS receives the SOAP request through its TCPIPSERVICE and looks up the matching URIMAP. The URIMAP instructs CICS that requests for the path /exampleApp/placeOrder are to use the WEBSERVICE placeOrder, the PIPELINE EXSECURE, the transaction ORDR and the user ID PRIVAT01. CICS starts pipeline processing using this information.

The context containers DFHWS-USERID and DFHWS-TRANID are initialized with the current user ID, PRIVAT01, and the current tranid, ORDR.

4. The pipeline contains the CICS-supplied security handler, which is configured to use an STS. The CICS-supplied security handler extracts the security credentials from the inbound SOAP request and sends a new SOAP request, through a special internal pipeline, to the STS requesting that new credentials are issued.
5. The STS service is provided by TFIM. TFIM validates the inbound UsernameToken (JOHNDOE/password), maps the user credentials to a different identity (USERWS05) and then issues a new UsernameToken for USERWS05.

Note: In our scenario, the UsernameToken issued by TFIM does not contain a password.

6. The CICS-supplied security handler checks the response from TFIM and updates the context container DFHWS-USERID with the new user ID, USERWS05.
7. The pipeline also includes the user-written message handler CIWSMSGO which sets the context container DFHWS-TRANID to contain ORDS for any placeOrder request.

CIWSMSGO is in this position in the pipeline for performance reasons. We authenticate the security token first and then switch the transaction ID. Otherwise, we would switch the transaction ID and then authenticate the security token. If the security token was invalid, then the transaction ID switch is an overhead.

8. As the context containers DFHWS-USERID and DFHWS-TRANID have been updated, and the terminal handler is one of the CICS-supplied SOAP handlers, the target application for placeOrder runs under user ID USERWS05 and tranid ORDS, as a new task.

10.2 Preparation for this scenario

In this section we provide a summary of the software used in the scenario and the main definitions used for the CICS, WebSphere and TFIM servers.

10.2.1 Software checklist

The software we used is listed in Table 10-1.

Table 10-1 Software used in the WS-Trust scenario

Windows	z/OS	TFIM
Internet Explorer V7.0	z/OS V1.9	TFIM Version 6.2.0.0
Windows XP Professional Version 2002 SP3	CICS Transaction Server V3.2	
IBM WebSphere Application Server V6.1.0.17		
IBM WebSphere Application Server Toolkit V6.1.1.6		
Our J2EE application <ul style="list-style-type: none"> ▶ ExampleAppClientV6.ear The Web client for the CICS catalog example application supplied with CICS TS. 	Our user-written CICS message handler programs <ul style="list-style-type: none"> ▶ CIWSMSGO This program changes the transaction ID for placeOrder requests to ORDS. ▶ SNIFFER (message handler program) This program browses through the containers available in the pipeline. 	

10.2.2 Definition checklist

The definitions we used are listed in Table 10-2.

Table 10-2 Settings used in the WS-Trust scenario

Value	CICS TS	TFIM	WebSphere Application Server
IP name	wtsc66.itso.ibm.com	mett.pdl.pok.ibm.com	cev8-pc7
TCP/IP port	14315	9610	9080
Jobname	CIWSS3C5		
APPLID	A6POS3C5		
TCPIPSERVICE	S3C5		
Provider PIPELINES	EXPIPE01 for inquireSingle and inquireCatalog services EXSECURE for placeOrder service		
Configuration files	ITSO_7658_basicso ap12provider.xml for inquireSingle and inquireCatalog services ITSO_7658_basicso ap12provider_with_STs.xml for placeOrder service		

The user IDs we used in our configuration are listed in Table 10-3.

Table 10-3 User IDs used in the WS-Trust scenario

Value	CICS TS
CICS region user ID	CIWS3D
User ID for which we wish to permit access to inquireSingle and inquireCatalog services	PUBLIC01
User ID for which we wish to permit access to the ORDR transaction of the placeOrder service	PRIVAT01
Requester's user ID (not known to RACF)	JOHNDOE

Value	CICS TS
Mapped user ID for which we want to permit access to the ORDS transaction of the placeOrder service	USERWS05

Because the Catalog application inquireSingle and inquireCatalog services are read-only services, whereas the placeOrder service updates the database, we show here how to secure the placeOrder service.

We run the inquireSingle and inquireCatalog services under a predefined user ID PUBLIC01 which is defined in inquireSingle and inquireCatalog wsbind files.

We document the following procedures to enable support for WS-Trust processing:

- ▶ Configuring the WebSphere service requester
- ▶ Configuring CICS for WS-Trust processing
- ▶ Configuring TFIM for WS-Trust processing
- ▶ Testing the WS-Trust scenario

10.3 Configuring the WebSphere service requester

In this section we show how to configure the client J2EE application to send SOAP messages with a security token. The steps to do this are:

- ▶ Import the ExampleAppClientV6 application to the Application Server Toolkit (AST).
- ▶ Configure the request generator to add a UsernameToken.
- ▶ Re-deploy the Web Service client application.

In the following sections we provide step-by-step details for each procedure.

Importing the base application

To configure our Web Service client, we used the IBM WebSphere Application Server Toolkit V6.1 (AST). We created a new workspace and imported the ExampleAppClientV6.ear file.

Configuring the request generator to add a UsernameToken

The AST provides two options to configure WS-Security:

- ▶ Enabling Web services security using the WS Security wizard.

This is a new feature in AST V6.1. You can enable production-level security for your Web services using the WS Security wizard, which provides the following options:

 - Add XML digital signature
 - Add XML encryption
 - Add a stand alone security token
- ▶ Enabling Web services security manually

If the WS Security wizard does not meet your needs, you have the option of manually securing Web services. This process gives you more control over the security settings, but is more time consuming and error prone than enabling security using the wizards.

We use the WS-Security wizard to configure the service client application by performing the following steps:

1. Expand **Web Services** in the Project Explorer and locate the service **DFH0XCMNService3**. This is the Web client service that places an order with the CICS catalog manager application. Right click the **DFH0XCMNService3** service and select **Secure Web Service Client > Add Stand Alone Security Token**, as shown in Figure 10-2.

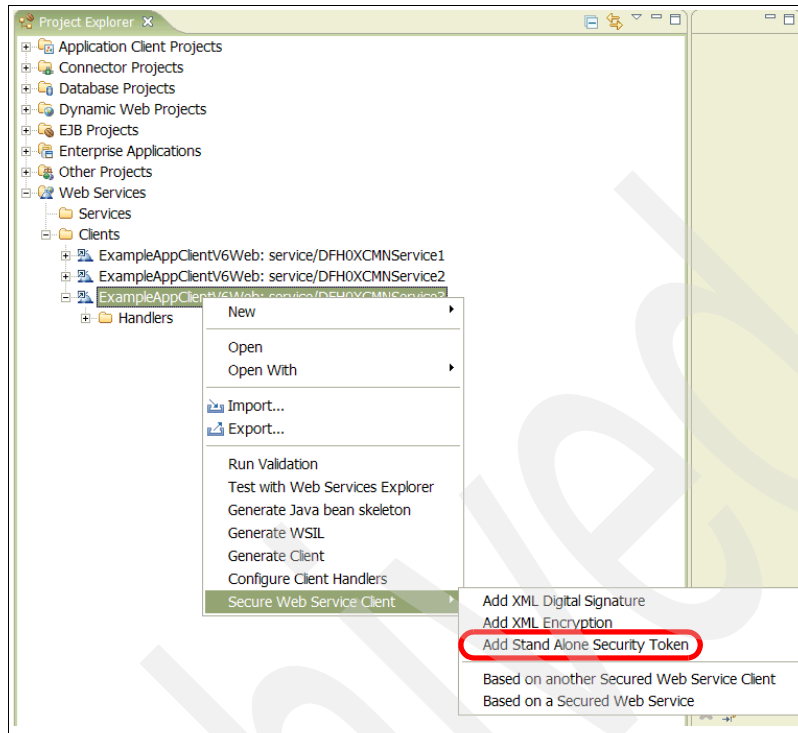


Figure 10-2 Application Server Toolkit WS-Security wizard

2. In the Add a Client Side Security Token page (Figure 10-3 on page 320) we accept the defaults:
 - Set the UsernameToken token type
 - Leave the URI blank
 - Leave the local name as the default for the token type
 - Leave the Callback handler as the NonPromptCallback handler

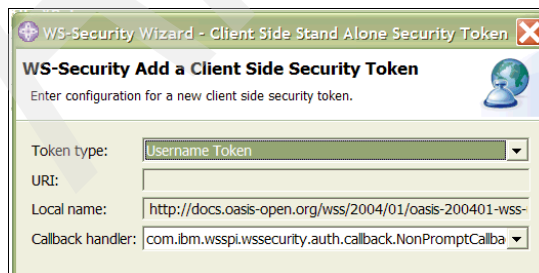


Figure 10-3 Application Server Toolkit WS-Security wizard - page 1

3. In the User Name and Password page, we specify the User ID and password which we want WebSphere Application Server to use to build the Username Token(Figure 10-4).

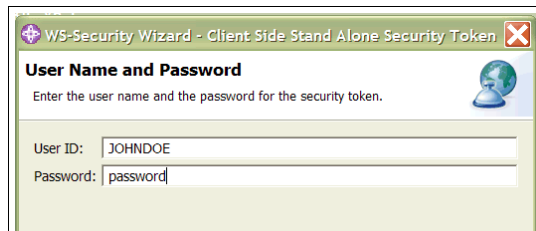


Figure 10-4 Application Server Toolkit WS-Security wizard - page 2

4. Click **Finish** and the WS-Security wizard creates a full set of deployment descriptor files. You can then review and edit these files using the Web Services editor by double-clicking on the **DFH0XCMNService3** service and opening the **WS Extension** and **WS Binding** tabs in the Web Deployment Descriptor.

Redeploying the Web service application

After configuring a signature security constraint for the service requester application, we exported a new EAR file called ExampleAppClientV6WSTrust.ear.

For further information about configuring WS-Security in WebSphere refer to *Web Services Handbook for WebSphere Application Server 6.1*, SG24-7257.

10.4 Configuring CICS for WS-Trust processing

In this section, we document the following steps:

- ▶ Enabling CICS to use WS-Trust support
- ▶ Defining a new pipeline to process placeOrder request messages using WS-Trust
- ▶ Authorising the service requester to run the placeOrder service

10.4.1 Enabling CICS to use WS-Trust support

To implement WS-Trust support, you must apply a number of updates to your CICS region. “Enabling CICS for WS-Security processing” on page 47 outlines the implementation pre-requisites.

On our system most of these pre-requisites were already in place, although we did need to add the CICSTS32.CICS.SDFHWSLD library to the DFHRPL concatenation.

10.4.2 Defining a new pipeline

Since the inquireCatalog and inquireSingle service requests will not be subject to WS-Trust processing, we need to create a new pipeline specifically for placeOrder requests.

Creating the pipeline configuration file

Example 10-1 shows the pipeline configuration file that we used for the WS-Trust test scenario. It includes the user-written handlers SNIFFER and CIWSMSGO, and configuration information for the CICS supplied security handler.

Example 10-1 Pipeline config file, ITSO_7658_basicsoap12provider_with_STS.xml

```
<?xml version="1.0" encoding="EBCDIC-CP-US"?>
<provider_pipeline
xmlns="http://www.ibm.com/software/http/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/http/cics/pipeline
provider.xsd ">
  <transport>
    <default_transport_handler_list>
      </default_transport_handler_list>
    </transport>
  <service>
    <service_handler_list>
      <handler>
        <program>SNIFFER</program>
        <handler_parameter_list/>
      </handler>
      <wsse_handler>
        <dfhwsse_configuration version="1">
          <sts_authentication action="issue">
            <auth_token_type>
              <namespace>http://docs.oasis-open.org/wss/2004/01/oasis-20
0401-wss-wssecurity-secext-1.0.xsd</namespace>
              <element>UsernameToken</element>
            </auth_token_type>
          </sts_authentication>
          <sts_endpoint>
            <endpoint>http://mett.pd1.pok.ibm.com:9610/TrustServer/Security
```

```

        TokenService</endpoint>
    </sts_endpoint>
</dfwsse_configuration>
</wsse_handler>
<handler>
    <program>CIWSMSG0</program>
    <handler_parameter_list/>
</handler>
<handler>
    <program>SNIFFER</program>
    <handler_parameter_list/>
</handler>
</service_handler_list>
<terminal_handler>
    <cics_soap_1.2_handler>
    </cics_soap_1.2_handler>
</terminal_handler>
</service>
<apphandler>DFHPITP</apphandler>
</provider_pipeline>

```

Tips: For details about which elements are required, optional or forbidden, in a pipeline configuration file see the CICS TS V3.2 Information Center.

It is possible to validate the pipeline config file XML against the correct schema (provider.xsd in this case) using an XML validation tool such as Rational Application Developer.

The `<wsse_handler>` element shown in Example 10-1 on page 322 contains a `<dfwsse_configuration>` element that specifies configuration information for the CICS-supplied security handler.

- ▶ In the `<sts_authentication>` element
 - The `action="issue"` attribute specifies that CICS is to request a security token from the STS.

Note: The attribute `action` is only valid in a provider pipeline. For a requester pipeline this attribute is assumed to be set to `issue`.

- ▶ `<auth_token_type>` is a child element of the `<sts_authentication>` element. It is used to determine which identity token CICS takes from the SOAP Header and sends to the STS. This element is mandatory for a requester pipeline but optional for a provider pipeline. If it is not specified for a provider pipeline then CICS will use the first identity token it recognizes in the message.

- <namespace> is a child element of the <auth_token_type> element. It is used together with the <element> element to select which token type should be selected by CICS.
- <element> is a child element of the <auth_token_type> element. It is used together with the <namespace> element to select which token type should be selected by CICS.

In the configuration file in Example 10-1 on page 322, the <namespace> and <element> values together indicate that a UsernameToken corresponding to the WS-Security specification is to be selected and sent to the STS.

- ▶ The <sts_endpoint> element contains one child element <endpoint>. The child element contains the URI that points to the location of the STS on the network. This can be HTTP, HTTPS or an MQ endpoint using the JMS format of URI.

Important: We specify an HTTP endpoint as an illustration only, and because we control the network over which the requests to the STS flow. We recommend that you use an SSL/ TLS connection to secure access to the STS.

The message handler program SNIFFER appears in the pipeline twice because we want to show the changes to the context containers made by the CICS security handler after the interaction with TFIM.

Creating a new pipeline to process placeOrder requests

We now create the new pipeline resource definition EXSECURE and we associate this pipeline with the configuration file ITSO_7658_basicoap12provider_with_STS.xml and the wsbind directory /CIWS/S3C5/wsbind/provider/secured/.

We do this using the RDO commands in Example 10-2.

Example 10-2 Create pipeline for WS-Trust scenario

```
CEDA COPY PIPELINE(EXPIPE01) GROUP(S3C5) AS(EXSECURE)
CEDA ALTER PIPELINE(EXSECURE) GROUP(S3C5)
Configfile(ITSO_7658_basicoap12provider_with_STS.xml)
Wsdir(/CIWS/S3C5/wsbind/provider/secured/)
```

Before installing the PIPELINE resource definition, we move the placeOrder.wsbind file to the /CIWS/S3C5/wsbind/provider/secured/ directory so that the placeOrder Web service is dynamically installed when the EXSECURE pipeline is installed.

10.4.3 Authorizing the service requester

In this section we show the RACF commands that are used to authorize the service requester to run the placeOrder Web service.

Permitting access

In Part 5.3, “Basic security configuration” on page 161 we show the RACF commands that we used to permit the pre-defined user ID (PRIVAT01) to start the ORDR transaction.

Example 10-3 shows the RACF commands that we used to permit the mapped user ID (USERWS05) to start the ORDS transaction.

Example 10-3 RACF command to allow USERWS05 to run ORDS transaction

```
PERMIT CIWS3D.ORDS CLASS(TCICSTRN) ID(USERWS05) ACCESS(READ)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

Permitting surrogate access

Example 10-4 shows the RACF commands that we used to permit the pre-defined user ID (PRIVAT01) to start transactions (such as ORDS) with the user ID USERWS05.

Example 10-4 RACF commands to allow PRIVAT01 to act as surrogate for USERWS05

```
RDEFINE SURROGAT USERWS05.DFHSTART UACC(NONE) OWNER(CICSR6)
PERMIT USERWS05.DFHSTART CLASS(SURROGAT) ID(PRIVAT01) ACCESS(READ)
SETROPTS RACLIST(SURROGAT) REFRESH
```

For more information regarding surrogate user security, see *CICS Transaction Server for z/OS RACF Security Guide*, SC34-6835.

10.5 Configuring TFIM for the WS-Trust scenario

In this section we show how to configure TFIM, in particular:

1. How to validate the inbound security token
2. How to map the user credentials to USERWS05
3. How to issue a new UsernameToken for USERWS05

We document the following steps:

- ▶ Removing the security constraint from the Trust Service

- ▶ Creating a Trust Chain to process STS requests from CICS

Full details of how to install, configure and manage Tivoli Federated Identity Manager can be found in the Tivoli Federated Identity Manager (TFIM) InfoCenter

http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/index.jsp?topic=/com.ibm.tivoli.fim.doc_6.2/welcome.htm

10.5.1 Removing the security constraint from the Trust Service

By default, the Trust Service provided by TFIM is protected by a security constraint; whether requests to TFIM arrive using HTTP or HTTPS, TFIM will reply with a HTTP 401 Authorization Required. CICS cannot reply with a user ID and password and so the request to the STS fails, as shown in Example 10-5.

Example 10-5 Error when TFIM requires Basic Authentication

```
DFHPI0504 07/16/2008 03:12:13 A6POS3C5 ORDR The CICS Pipeline Manager
has failed to communicate with a remote server due to an error
          in the underlying transport. TRANSPORT: HTTP, PIPELINE:
DFHPITCR.
```

```
DFHPI0512 07/16/2008 03:12:13 A6POS3C5 ORDR The CICS Pipeline Manager
has received a fault from the target Secure Token Service:
```

```
http://mett.pdl.pok.ibm.com:9610/TrustServer/SecurityTokenService. The
fault had a fault code of SOAP-ENV:Server.
```

There are two options in TFIM to disable authorization checking by WebSphere based on the security constraints defined for the STS:

- ▶ Disable WebSphere Application Security
- ▶ Modify the Web deployment descriptor for the STS Web module

We chose to modify the Web deployment descriptor for the STS Web module.

The security constraints for the TFIM Security Token Service Web Module are defined in the Web deployment descriptor file (web.xml).

The file

WebSphere_Installation_Directory/profiles/profile/config/cells/cell/applications/ITFIMRuntime.ear/deployments/ITFIMRuntime/com.tivoli.am.fim.war.sts.war/WEB-INF/web.xml can be edited with any XML editor.

Attention: Care must be taken when editing this file.

The file contains a number of “security constraints” which require clients of the STS to send a user ID and password using HTTP Basic Authentication. We remove the security constraints by following the steps below.

1. Remove the security constraint relating to the URL /SecurityTokenService, as shown in Example 10-6.

Example 10-6 First security constraint to be removed from TFIM web.xml file

```
<security-constraint>
  <display-name>TrustServerPortConstraint</display-name>
  <web-resource-collection>
    <web-resource-name>TrustServerPort</web-resource-name>
    <url-pattern>/SecurityTokenService</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>TrustServerSecurity:::Internal Users allowed access to
      the TrustServer</description>
    <role-name>TrustClientInternalRole</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

2. Remove the security constraint for the URL /*, as shown in Example 10-7.

Example 10-7 Second security constraint to be removed from TFIM web.xml file

```
<security-constraint>
  <display-name>DenyAccessConstraint</display-name>
  <web-resource-collection>
    <web-resource-name>Default Deny</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description>Default Deny</description>
    <role-name>FIMNobody</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

3. Remove the security constraint for the role TrustClientInternalRole, as shown in Example 10-8.

Example 10-8 Third security constraint to be removed from TFIM web.xml file

```
<security-role>
  <description>Internal Users allowed access to the
TrustServer</description>
  <role-name>TrustClientInternalRole</role-name>
</security-role>
```

4. Remove the security constraint for the FIMNobody role, as shown in Example 10-9.

Example 10-9 Fourth security constraint to be removed from TFIM web.xml file

```
<security-role>
  <description>Role indicating no access to anyone</description>
  <role-name>FIMNobody</role-name>
</security-role>
```

5. Restart the TFIM Server for these changes to take effect.

10.5.2 Creating a Trust Chain to process STS requests from CICS

In this section we show how to configure TFIM to provide a Security Token Service by using a Trust Chain to validate the inbound security token, to map the user credentials to another user ID and to issue a new UsernameToken.

We document the following steps:

- ▶ Logging on to the TFIM Integrated Solutions Console
- ▶ Creating a Trust Chain
- ▶ Configuring the Trust Chain with validate, mapping and issue modules

Logging on to the TFIM Integrated Solutions Console

TFIM runs as a WebSphere Application Server application and is administered by the WebSphere Integrated Solutions Console. We use the following URL to access our TFIM server:

<http://mett.pdl.pok.ibm.com:9510/ibm/console>

After logging on to the admin console, we are presented with the screen shown in Figure 10-5 on page 329. Highlighted in Figure 10-5 are the TFIM Version Number 6.2.0.0 and, in the left hand panel of the admin console, the tab for administering TFIM.

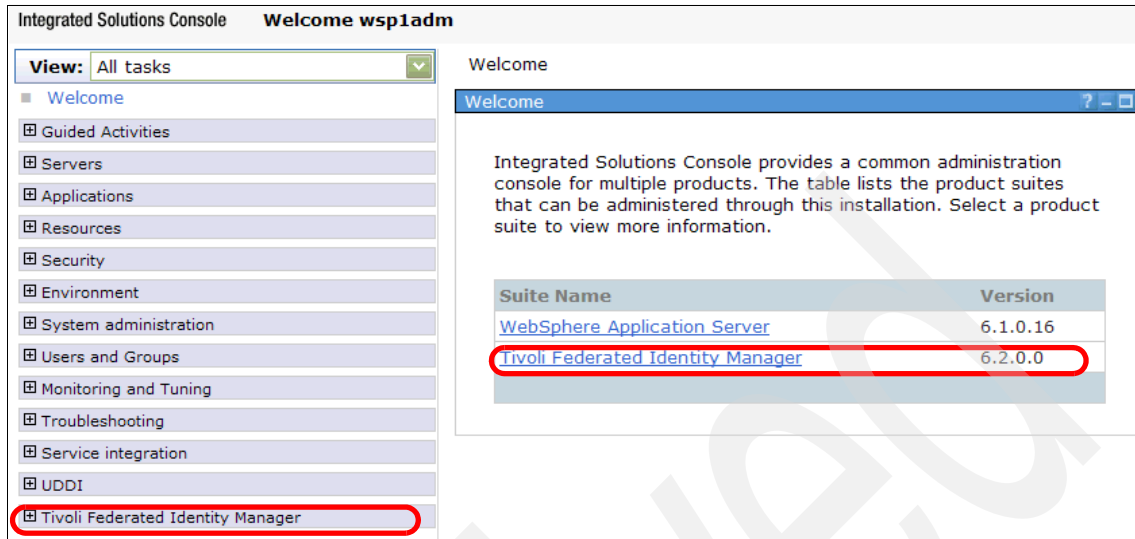


Figure 10-5 Integrated Solutions Console of TFIM, highlighting TFIM tasks in menu

Creating a Trust Chain

A Trust Chain can be considered as analogous to a pipeline in CICS. It consists of a number of pre-defined programs, executed in a pre-defined order, with parameters passed to them at runtime.

We require a Trust Chain in TFIM to perform three tasks:

- ▶ Validate the incoming UsernameToken of JOHND0E/password to ensure that the user ID JOHND0E exists and that the correct password has been used.
- ▶ Map the user ID JOHND0E to a new user ID, USERWS05, that is recognized by RACF.
- ▶ Issue a new UsernameToken which is returned to CICS. The user ID associated with the UsernameToken is USERWS05. The UsernameToken issued by TFIM does not include a password.

To do this, we perform the following steps:

1. Expand **Tivoli Federated Identity Manager** in the left column of the Integrated Solutions Console.
2. Expand **Configure Trust Service**.

3. Select **Trust Service Chains** as shown in Figure 10-6.



Figure 10-6 TFIM Integrated Solutions Console Configure Trust Service menu

4. On the Trust Service Chains screen, select **Create...** as shown in Figure 10-7.

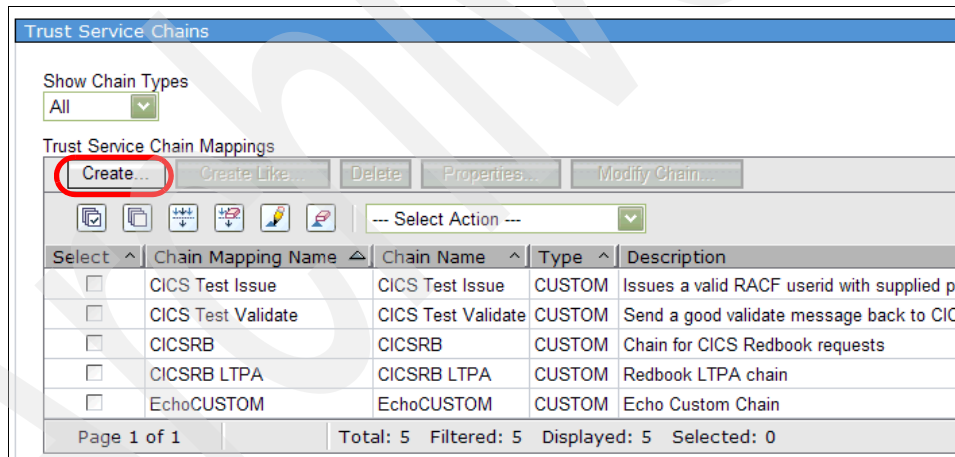


Figure 10-7 Create Trust Service Chains on the Trust Service Chains screen

5. This starts the Trust Service Chain Mapping Wizard which assists in the creation of Trust Service Chains, as shown in Figure 10-8. Click **Next**.

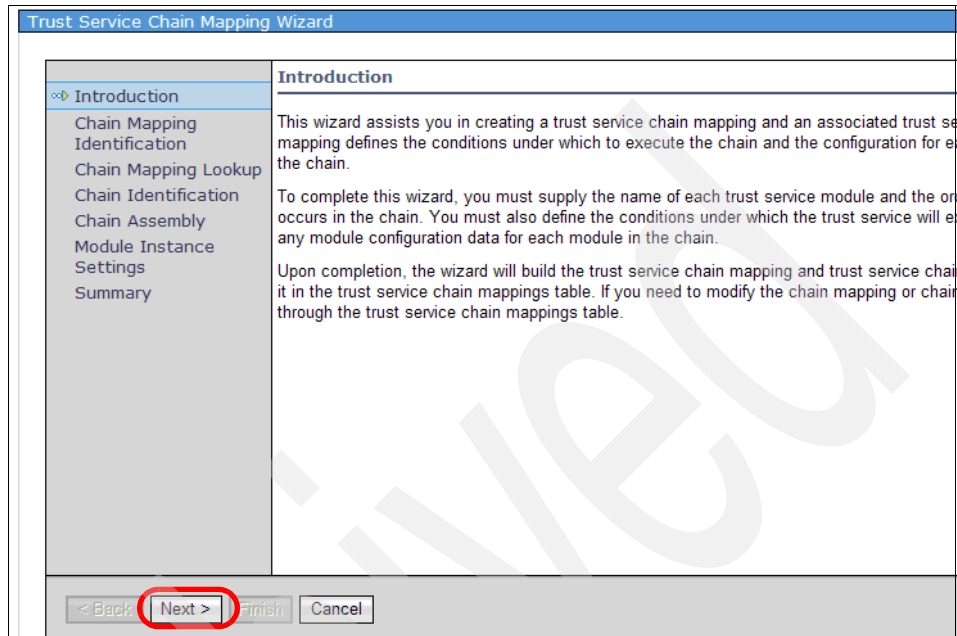


Figure 10-8 Trust Service Chain Mapping Wizard

6. The Chain Mapping Identification screen shown in Figure 10-9 requires us to specify a name for our Trust Service Chain. We choose CICS WS-Security Redbook.

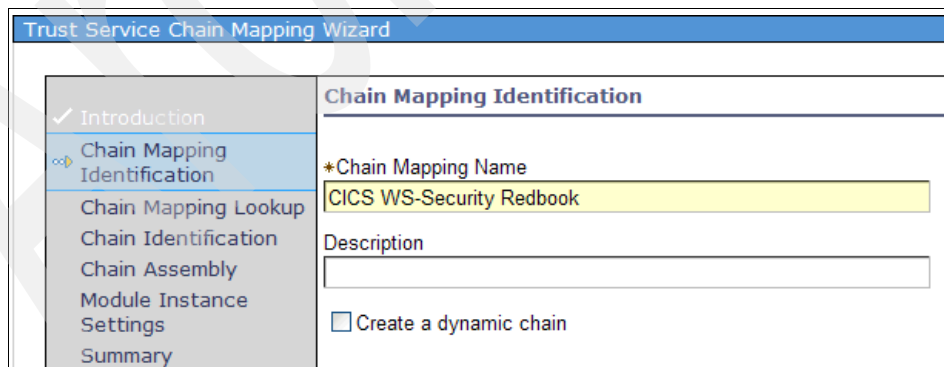


Figure 10-9 Chain Mapping Identification in Chain Mapping Wizard

7. The Chain Mapping Lookup screen shown in Figure 10-10 on page 332 determines when the Trust Service Chain is used. CICS makes Issue requests of TFIM so we select the Issue **Request Type**. We choose to use **Traditional WS-Trust Elements**. We then use the **AppliesTo** field so that this chain is only used for requests that apply to the URI /exampleApp*. This is done by using a Regular Expression®, which is entered as REGEXP:(./exampleApp.*). Finally we select **Token Type** of Any token, so that requests for any token type can be processed by this Chain.

Chain Mapping Lookup

Request Type

Request Type: Request Type URI:

Lookup Type

Use Traditional WS-Trust Elements (AppliesTo, Issuer, and TokenType)
 Use XPath to Define Custom Lookup Rule

AppliesTo

Address:
Service Name: :
Port Type: :

Issuer

Address:
Service Name: :
Port Type: :

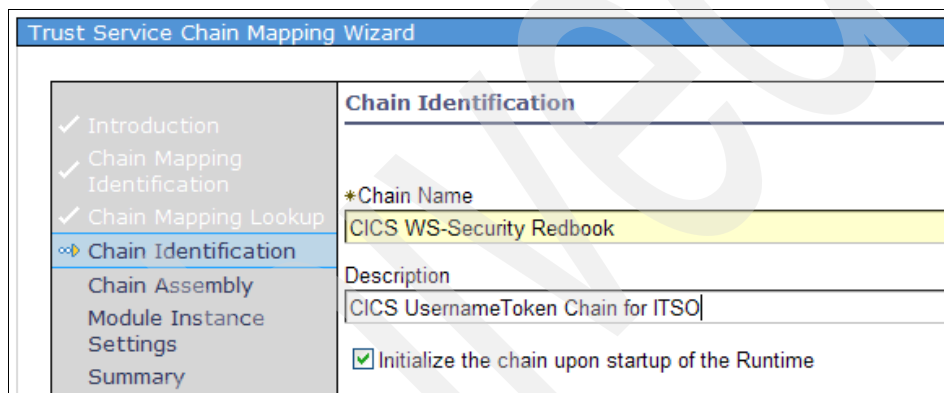
Token Type

Token Type: Token Type URI:

Figure 10-10 Chain Mapping Lookup screen of Chain Mapping Wizard

Tip: AppliesTo will accept a * as the last character of the string. For our application, we could code /exampleApp/placeOrder* and this would also match. For more complicated wild-carded URIs, regular expressions as shown above must be used.

- The next screen (Figure 10-11) is the Chain Identification Screen. The Chain Name is already filled in with CICS WS-Security Redbook, so we add a description and select the **Initialize the chain upon startup of the Runtime** option. This means that whenever TFIM starts, the CICS WS-Security Redbook chain will also start.



The screenshot shows the 'Trust Service Chain Mapping Wizard' with the 'Chain Identification' step selected. The wizard has a sidebar with the following steps: Introduction, Chain Mapping Identification, Chain Mapping Lookup, Chain Identification (selected), Chain Assembly, Module Instance Settings, and Summary. The main area is titled 'Chain Identification' and contains the following fields:

*Chain Name	CICS WS-Security Redbook
Description	CICS UsernameToken Chain for ITSO
<input checked="" type="checkbox"/> Initialize the chain upon startup of the Runtime	

Figure 10-11 Chain Identification screen of Chain Mapping Wizard

- The next screen (Figure 10-12) is the Chain Assembly screen. This screen allows the addition of modules to the Trust Chain. Modules can be considered somewhat analogous to CICS pipeline handlers. We see that the chain assembly is initially empty.

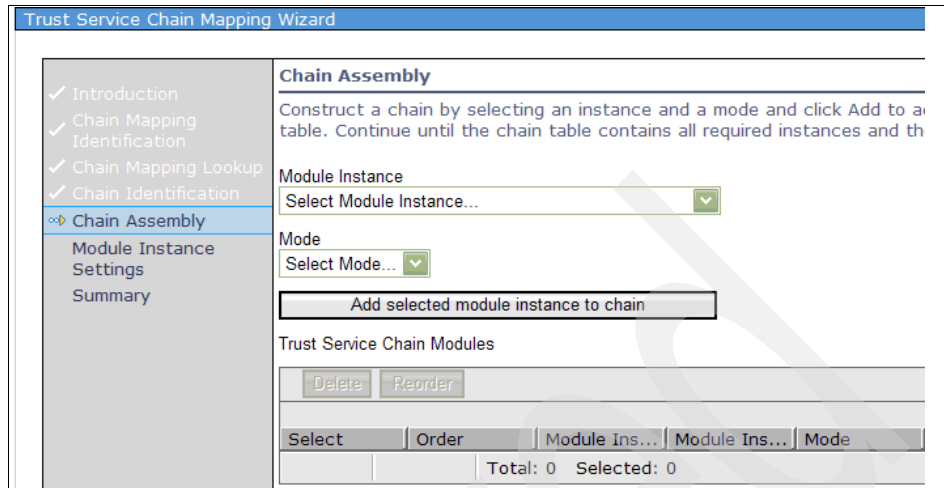


Figure 10-12 Chain Assembly screen of Chain Mapping Wizard Part 1 of 4

10. We select a Module Instance of Default UsernameToken and a Mode of Validate, and then click **Add selected module instance to chain**. The results are shown in Figure 10-13.

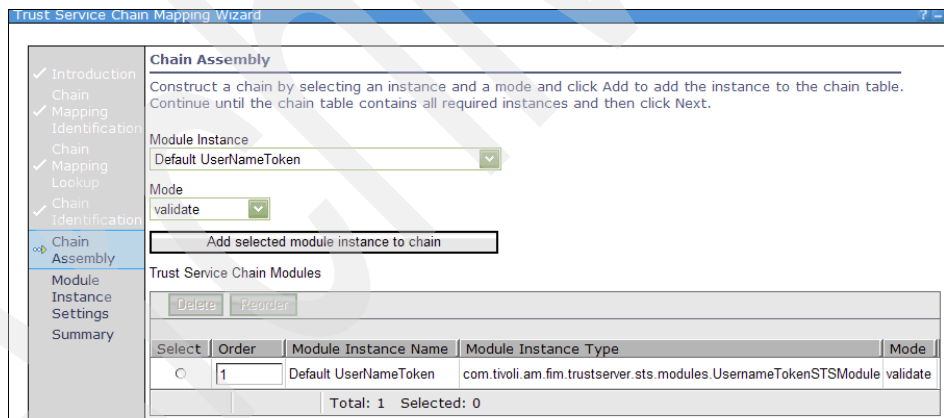


Figure 10-13 Chain Assembly screen of Chain Mapping Wizard Part 2 of 4

11. We select a Module Instance of Default Map Module. The Default Map Module has only one possible Mode, so we accept the default of map. We click **Add selected module instance to chain**. The results are shown in Figure 10-14.

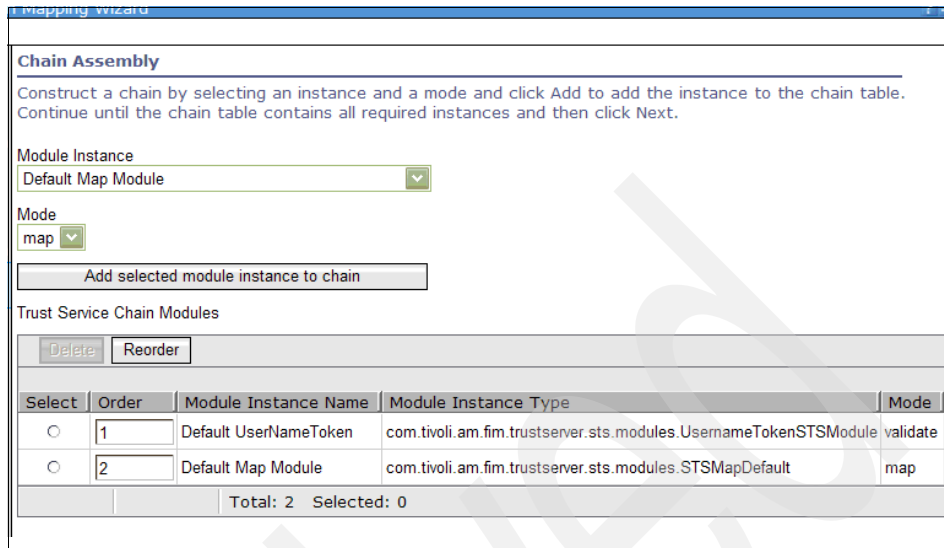


Figure 10-14 Chain Assembly screen of Chain Mapping Wizard Part 3 of 4

12. We select a Module Instance of Default UserNameToken and a Mode of Issue, and then click **Add selected module instance to chain**. The results are shown in Figure 10-15.

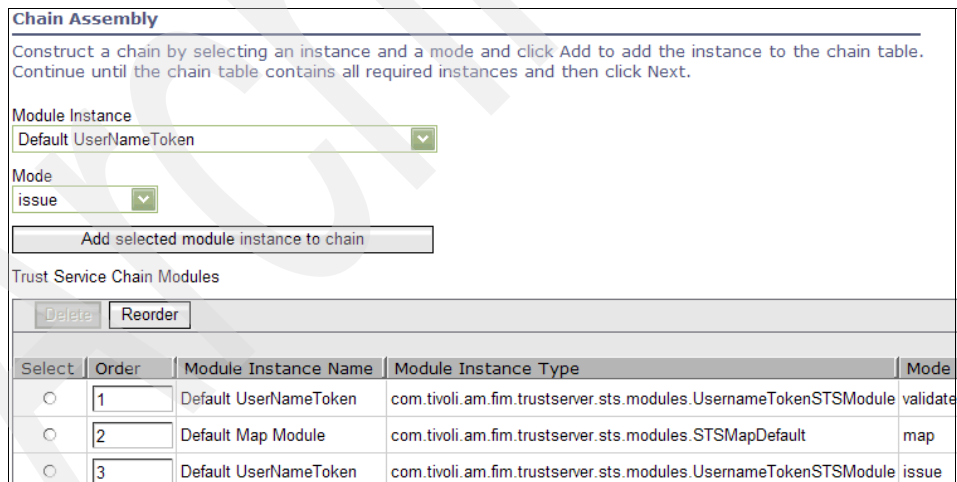


Figure 10-15 Chain Assembly screen of Chain Mapping Wizard Part 4 of 4

13. TFIM now asks us to configure each of our three modules. The first module is the Default UsernameToken module in Validate mode. We configure this module as shown in Figure 10-16.

- We ensure that the **Skip password validation** check box is clear because we want TFIM to validate the password of any Username Token we send to the STS.
- We select **Use WebSphere Registry for authentication**. This is because our TFIM administrator has configured WebSphere Registry with the security credentials used for this scenario. Other options are to use Tivoli Access Manager or to use JAAS (Java Authentication and Authorization Service).
- Finally, under **Partner Properties**, we set **Amount of time the token is valid after being issued (seconds)** to -1. This means that no timestamp is required or validated in the inbound security token. Setting this to a positive value causes TFIM to check the timestamp of the token and reject tokens that are too old. This could be used to prevent replay attacks.

The screenshot shows the 'Username Token Module Configuration' screen. At the top, it says 'Enter the required values to configure the Username Token Module.' Below this, there are two sections: 'Federation properties' and 'Partner properties'. In the 'Federation properties' section, there is a checkbox for 'Skip password validation' which is unchecked. Below it are three radio buttons for authentication options: 'Use Tivoli Access Manager for authentication' (unchecked), 'Use WebSphere Registry for authentication' (checked), and 'Use JAAS for authentication' (unchecked). There is also a field for '*JAAS Login Context Alias' with the value 'WSLogin'. Below that are empty text boxes for 'JAAS Provider host name' and 'JAAS Provider port'. In the 'Partner properties' section, there is a field for '*Amount of time the token is valid after being issued (seconds)' with the value '-1'.

Figure 10-16 Username Token Module Configuration screen for Mode (Validate)

14. The second module is the Default Mapping module. This is configured by way of XSLT, and TFIM needs to know the location of the XSLT file, as shown in Figure 10-17.

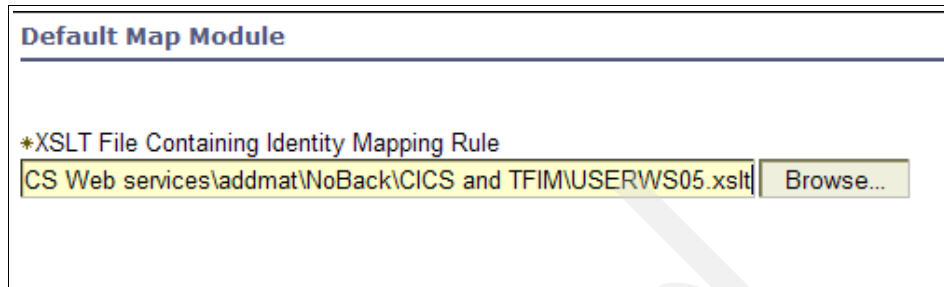


Figure 10-17 Default Map Module screen

15. The contents of the XSLT file are shown in Example 10-10. For this example we have chosen to map any valid user ID to USERWS05. Other mappings are possible using XSLT, for example one-to-one or many-to-one mappings.

Example 10-10 XSLT file used for mapping to USERWS05

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0" xmlns:stsuuser="urn:ibm:names:ITFIM:1.0:stsuuser">
<xsl:strip-space elements="*" />
<xsl:output method="xml" version="1.0" encoding="utf-8" indent="yes" />
<xsl:template match="@* | node()">
  <xsl:copy>
    <xsl:apply-templates select="@* | node()" />
  </xsl:copy>
</xsl:template>
<xsl:template
  match="//stsuuser:Principal/stsuuser:Attribute[@name='name']">
  <stsuuser:Attribute name="name"
type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
    <stsuuser:Value>USERWS05</stsuuser:Value>
  </stsuuser:Attribute>
</xsl:template>
<xsl:template
match="//stsuuser:Principal/stsuuser:Attribute[@name='Username']">
  <stsuuser:Attribute name="Username"
type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
    <stsuuser:Value>USERWS05</stsuuser:Value>
  </stsuuser:Attribute>
</xsl:template>
</xsl:stylesheet>

```

The mapping rules are described in eXtensible Stylesheet Language Text format (XSLT). XSLT is used to transform one piece of XML into another. TFIM calls the Mapping Module with XML based on the incoming security token, but converted into a token-neutral format. This means that the same mapping rules can be used for LTPA tokens, UsernameTokens or Kerberos Tickets.

XSLT works on templates, `<xsl:template>`, as shown above. Templates have an attribute of *match* which determines which part of the XML document they apply to. If an XML document contains an element which matches a template, then processing and transformation of the XML occurs.

The XSLT shown in Example 10-10 does the following:

- ▶ The first highlighted section copies all the XML from the incoming message to the outgoing message, unless it is matched elsewhere. The default for XSLT is to discard all XML that is not matched; this rule ensures that all XML is copied or changed.
- ▶ The second highlighted section matches XML with the structure shown in Example 10-11:

Example 10-11 Approximation of Security token service universal user XML (input)

```
<stsuser:Principal>
  <stsuser:Attribute name="Password"
type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd#PasswordText">
  <stsuser:Value>*****</stsuser:Value>
</stsuser:Attribute>
  <stsuser:Attribute name="Username"
type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
  <stsuser:Value>JOHNDOE</stsuser:Value>
</stsuser:Attribute>
  <stsuser:Attribute name="name"
type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
  <stsuser:Value>JOHNDOE</stsuser:Value>
</stsuser:Attribute>
</stsuser:Principal>
```

- ▶ The XSLT then changes the content of <stsuser:Value> to USERWS05, as shown in Example 10-12.

Example 10-12 Approximation of Security token service universal user XML (output)

```
<stsuser:Principal>
  <stsuser:Attribute name="Password"
type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd#PasswordText">
  <stsuser:Value>*****</stsuser:Value>
</stsuser:Attribute>
  <stsuser:Attribute name="Username"
type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
  <stsuser:Value>USERWS05</stsuser:Value>
</stsuser:Attribute>
  <stsuser:Attribute name="name"
type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
  <stsuser:Value>USERWS05</stsuser:Value>
</stsuser:Attribute>
</stsuser:Principal>
```

16. The final module is the Default UsernameToken module, for mode Issue. TFIM offers us various options for issuing a new UsernameToken. We accept the defaults, which are:
- **Include nonce in token.** Create a number-used-once (nonce) to prevent replay attacks. CICS tolerates this but does not use it.
 - **Include timestamp in token.** Mark the returned token with a timestamp to prevent replay attacks. CICS tolerates this but does not use it.
 - **Do not include the password.** We do not want a password returned to CICS.

The Username Token Module configuration is shown in Figure 10-18.

Username Token Module Configuration

Enter the required values to configure the Username Token Module.

Federation properties

Include nonce in token

Include token creation time in token

Partner properties

Options for including password in the token

Do not include the password

Include the digest of the password value

Include the password in clear text

Figure 10-18 Username Token Module Configuration for Mode (Issue)

17. TFIM presents us with a summary screen, as shown in Figure 10-19.

Summary

Chain Mapping Identification

Chain Mapping Name: CICS WS-Security Redbook

Description:

Create a dynamic chain: false

Chain Mapping Lookup

Request Type

Request Type: http://schemas.xmlsoap.org/ws/2005/02/trust/Issue

AppliesTo

Address: /exampleApp/placeOrder

Figure 10-19 Trust Service Chain Wizard Summary

Click **Finish**.

18. The wizard completes, and TFIM warns us that **recent configuration changes need to be reloaded** (Figure 10-20).

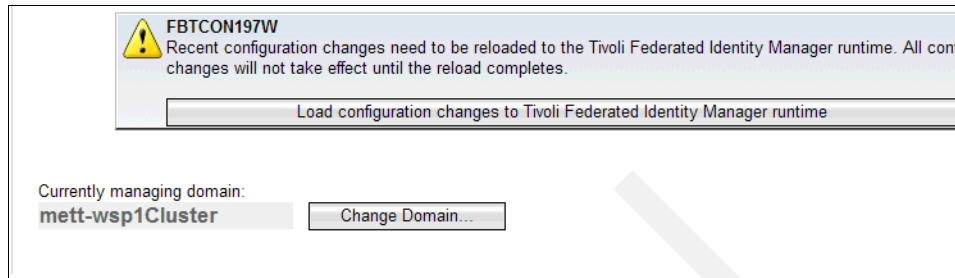


Figure 10-20 FBTCOM197W Warning Message from TFIM

We complete the dialog by clicking **Load configuration changes to Tivoli Federated Identity Manager runtime**.

10.6 Testing the WS-Trust scenario

In order to test our WS-Trust scenario, we deploy the configured Web client application (file **ExampleAppClientV6WSTrust.ear**) to our WebSphere Application Server running on Windows. See “Installing and setting up the application” on page 174 for more information on deploying and testing the sample application.

We submit a placeOrder request so that a Web service request is sent from WebSphere Application Server to CICS.

In this section we show the results of testing the scenario. We show these tasks:

- ▶ Request from WebSphere to CICS captured with TCP/IP Monitor
- ▶ Request from CICS to TFIM captured with CICS PI Level 2 Trace
- ▶ Response from TFIM to CICS captured with CICS PI Level 2 Trace
- ▶ CEMT showing that the tasks run under the correct user IDs
- ▶ Output from SNIFFER showing context-switching

Request from WebSphere to CICS

An example of the request message with UsernameToken sent by WebSphere is shown in Example 10-13.

Example 10-13 SOAP Request message with Username Token

```
<soapenv:Envelope
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
```

```

xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Header>
<wsse:Security
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wss
ecurity-secext-1.0.xsd" soapenv:mustUnderstand="1">
<wsse:UsernameToken>
<wsse:Username>JOHNDOE</wsse:Username>
<wsse:Password
Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-
token-profile-1.0#PasswordText">password</wsse:Password>
</wsse:UsernameToken>
</wsse:Security>
</soapenv:Header>
<soapenv:Body>
<p635:DFHOXCMNOperation
xmlns:p635="http://www.DFHOXCMN.DFHOXCP5.Request.com">
<p635:ca_request_id>01ORDR</p635:ca_request_id>
<p635:ca_return_code>0</p635:ca_return_code>
<p635:ca_response_message>[C@6e686e68</p635:ca_response_message>
<p635:ca_order_request>
<p635:ca_user ID>John Doe</p635:ca_user ID>
<p635:ca_charge_dept>123</p635:ca_charge_dept>
<p635:ca_item_ref_number>10</p635:ca_item_ref_number>
<p635:ca_quantity_req>1</p635:ca_quantity_req>
<p635:filler1 xsi:nil="true"/>
</p635:ca_order_request>
</p635:DFHOXCMNOperation>
</soapenv:Body>
</soapenv:Envelope>

```

-
- ▶ The <wsse:Security> header includes the mustUnderstand="1" attribute, which indicates that either this header must be processed or a SOAP fault thrown.
 - ▶ The <wsse:UsernameToken> contains the security credentials that we pass into CICS. There are two child elements.
 - The <wsse:Username> element contains John Doe's user name, which is JOHNDOE.
 - The <wsse:Password> element has an attribute, Type, which refers to the form in which the password is coded. There are two types defined in the Username Token specification, text and digest. We use text.

Request from CICS to TFIM

The WS-Trust handler within the pipeline makes a RequestSecurityToken request to TFIM. An example of this message is shown in Example 10-14.

Example 10-14 RequestSecurityToken request sent to TFIM

```
<SOAP-ENV:Envelope
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<wst:RequestSecurityToken
xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wssc="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
<wst:RequestType>
http://schemas.xmlsoap.org/ws/2005/02/trust/Issue
</wst:RequestType>
<wst:TokenType>
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd/UsernameToken
</wst:TokenType>
<wst:Base><wsse:UsernameToken>
<wsse:Username>JOHNDOE</wsse:Username>
<wsse:Password
Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">password</wsse:Password>
<wsu:Created
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">2008-07-18T08:38:54+00:00</wsu:Created></wsse:UsernameToken>
</wst:Base>
<wsp:AppliesTo><wsa:EndpointReference>
<wsa:Address>/exampleApp/placeOrder/DFHOXCMNOperation</wsa:Address>
</wsa:EndpointReference></wsp:AppliesTo>
</wst:RequestSecurityToken>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

- ▶ The SOAP Body contains the <wst:RequestSecurityToken> element which contains a number of child elements.
 - <wst:RequestType> contains the type of the request from the WS-Trust Specification. This is an *issue* request.
 - <wst:TokenType> contains the type of token that CICS expects TFIM to return. For CICS as a Web Services provider, this token is always UsernameToken. For CICS as a Web Services requester, the token type is specified in <auth_token_type> in the pipeline config file.
 - <wst:Base> contains the UsernameToken that CICS has extracted from the incoming message. We see John Doe's user name, which is **JOHNDOE**, and his password, which is **password**.
 - <wsp:AppliesTo> contains a child element <wsa:EndpointReference> which has a child element <wsa:Address>. This contains the URI of the Web Service that John Doe is trying to access.

Note: the Address used is made up of the URI from the URIMAP and the Operation from the Web Service. This is why we use a Regular Expression, /exampleApp* so TFIM will match this.

Response from TFIM to CICS

TFIM processes the RequestSecurityToken and sends back a RequestSecurityTokenResponse. An example of this message is shown in Example 10-15.

Example 10-15 RequestSecurityTokenResponse received from TFIM

```

<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Header/>
<soapenv:Body>
<wst:RequestSecurityTokenResponse Context=""
wsu:Id="uuid35525a7b-011b-1579-a88c-a471075f6d27"
xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
curity-utility-1.0.xsd">
<wst:TokenType>http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd/UsernameToken</wst:TokenType>
<wsp:AppliesTo
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">

```



```

<wsa:EndpointReference>
<wsa:Address>/exampleApp/placeOrder/DFHOXCMNOperation</wsa:Address>
</wsa:EndpointReference>
</wsp:AppliesTo>
<wst:RequestedSecurityToken>
<wss:UsernameToken
wsu:Id="username35525a90-011b-1d3d-ba79-a471075f6d27"
xmlns:wss="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
curity-secext-1.0.xsd"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
curity-utility-1.0.xsd">
<wss:Username>USERWS05</wss:Username>
<wss:Nonce
EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-s
oap-message-security-1.0#Base64Binary">
wYrYthvUfNBGJfB2qG6/YQ==
</wss:Nonce>
<wsu:Created>2008-07-18T08:38:54+00:00</wsu:Created>
</wss:UsernameToken>
</wst:RequestedSecurityToken>
<wst:RequestedAttachedReference
xmlns:wss="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
curity-secext-1.0.xsd">
<wss:SecurityTokenReference>
<wss:Reference URI="#username35525a90-011b-1d3d-ba79-a471075f6d27"
ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-user
name-token-profile-1.0#UsernameToken"
xmlns:wss="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
curity-secext-1.0.xsd"/>
</wss:SecurityTokenReference>
</wst:RequestedAttachedReference>
</wst:RequestSecurityTokenResponse>
</soapenv:Body>
</soapenv:Envelope>

```

-
- ▶ The SOAP Body contains the <wst:RequestSecurityTokenResponse> element which contains two child elements, <wst:RequestedSecurityToken> and <wst:RequestedAttachedReference>.
 - <wst:RequestedSecurityToken> contains the token returned from TFIM.
 - <wss:UsernameToken> is the immediate child element of <wst:RequestedSecurityToken> and contains three child elements:
 - <wss:Username> contains the mapped user ID **USERWS05**
 - <wss:Nonce> is an element tolerated by CICS but not used.

- <wss:Created> is an element tolerated by CICS but not used.
- <wst:RequestedAttachedReference> contains the schema definitions of the UsernameToken.

CEMT INQUIRE TASK

Example 10-16 shows the ORDS transaction running with the user ID USERWS05 (the user ID returned from TFIM), while the ORDR transaction continues to execute with the PRIVAT01 user ID.

Example 10-16 ORDS executing with user ID returned from TFIM

```

INQUIRE TASK
STATUS: RESULTS - OVERTYPE TO MODIFY
Tas(0000049) Tra(ORDR)          Sus Tas Pri( 001 )
  Sta(U ) Use(PRIVAT01) Uow(C2ABAAF9240CED04) Hty(RZCBNOTI)
Tas(0000050) Tra(ORDS)          Sus Tas Pri( 001 )
  Sta(U ) Use(USERWS05) Uow(C2ABAAF95DCC6B4B) Hty(EDF )
Tas(0000052) Tra(CEDF) Fac(E024) Sus Ter Pri( 001 )
  Sta(SD) Use(CICRSR8 ) Uow(C2ABAAF9704F3349) Hty(ZCIOWAIT)

```

SNIFFER Output showing context-switching

The output from the SNIFFER program is detailed in Example 10-17 and Example 10-18.

Example 10-17 Output from SNIFFER during RECEIVE-REQUEST processing

```

1  ORDR 20080710081658 SNIFFER : *** Start *** RECEIVE-REQUEST
   ORDR 20080710081658 SNIFFER : >=====
   ORDR 20080710081658 SNIFFER : Container Name   : DFHFUNCTION
   ORDR 20080710081658 SNIFFER : Content length  : 00000016
   ORDR 20080710081658 SNIFFER : Container content: RECEIVE-REQUEST
   ORDR 20080710081658 SNIFFER : >=====
   ORDR 20080710081658 SNIFFER : Container Name   : DFHWS-SOAPACTION
   ORDR 20080710081658 SNIFFER : Content length  : 00000002
   ORDR 20080710081658 SNIFFER : Container content: ""
   ORDR 20080710081658 SNIFFER : >=====
   ORDR 20080710081658 SNIFFER : Container Name   : DFHWS-URI
   ORDR 20080710081658 SNIFFER : Content length  : 00000022
   ORDR 20080710081658 SNIFFER : Container content: /exampleApp/placeOrder
   ORDR 20080710081658 SNIFFER : >=====
   ORDR 20080710081658 SNIFFER : Container Name   : DFHREQUEST
   ORDR 20080710081658 SNIFFER : Content length  : 00001199
   ORDR 20080710081658 SNIFFER : Container content: <soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"

```

```

xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Header><wsse:Security soapenv:mustUnderstand="1"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
<wsse:UsernameToken>
<wsse:Username>JOHNDOE</wsse:Username><wsse:Password
Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">password</wsse:Password>
</wsse:UsernameToken>
</wsse:Security></soapenv:Header><soapenv:Body><p635:DFHOXCMNOperation
xmlns:p635="http://www.DFHOXCMN.DFHOXCP5.Request.com"><p635:ca_request_id>010RDR</p635:ca_request_id><p635:ca_return_code>0</p635:ca_return_code><p635:ca_response_message>^C@1280128</p635:ca_response_message><p635:ca_order_request><p635:ca_userid>a</p635:ca_userid><p635:ca_charge_dept>a</p635:ca_charge_dept><p635:ca_item_ref_number>10</p635:ca_item_ref_number><p635:ca_quantity_req>1</p635:ca_quantity_req><p635:filler1 xsi:nil="true"/></p635:ca_order_request></p635:DFHOXCMNOperation></soapenv:Body></soapenv:Envelope>
  ORDR 20080710081658 SNIFFER : >=====
  ORDR 20080710081658 SNIFFER : Container Name : DFHWS-PIPELINE
  ORDR 20080710081658 SNIFFER : Content length : 00000008
  ORDR 20080710081658 SNIFFER : Container content: EXSECURE
  ORDR 20080710081658 SNIFFER : >=====
  ORDR 20080710081658 SNIFFER : Container Name : DFHWS-USERID
  ORDR 20080710081658 SNIFFER : Content length : 00000008
  ORDR 20080710081658 SNIFFER : Container content: PRIVATO1
  ORDR 20080710081658 SNIFFER : >=====
  ORDR 20080710081658 SNIFFER : Container Name : DFHWS-TRANID
  ORDR 20080710081658 SNIFFER : Content length : 00000004
  ORDR 20080710081658 SNIFFER : Container content: ORDR
  ORDR 20080710081658 CIWSMSG0: >=====
  ORDR 20080710081658 CIWSMSG0: Container Name: : DFHWS-WEBSERVICE
  ORDR 20080710081658 CIWSMSG0: Container content: placeOrder
  ORDR 20080710081658 CIWSMSG0: -----
  ORDR 20080710081658 CIWSMSG0: Container Name: : DFHWS-TRANID
  ORDR 20080710081658 CIWSMSG0: Container content: ORDS
  ORDR 20080710081658 CIWSMSG0: -----
  ORDR 20080710081658 CIWSMSG0: Container Name: : DFHWS-URI
  ORDR 20080710081658 CIWSMSG0: Container content: /exampleApp/placeOrder
  ORDR 20080710081658 SNIFFER : >=====
  ORDR 20080710081658 SNIFFER : Container Name : DFHWS-XMLNS
  ORDR 20080710081658 SNIFFER : Content length : 00000366
  ORDR 20080710081658 SNIFFER : Container content:
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"

```

```

xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  ORDER 20080710081658 SNIFFER : >=====
  ORDER 20080710081658 SNIFFER : Container Name : DFHWS-ST5-EP
  ORDER 20080710081658 SNIFFER : Content length : 00000065
  ORDER 20080710081658 SNIFFER : Container content:
http://mett.pdl.pok.ibm.com:9610/TrustServer/SecurityTokenService
  ORDER 20080710081658 SNIFFER : >=====
  ORDER 20080710081658 SNIFFER : Container Name : DFHWS-SIG
  ORDER 20080710081658 SNIFFER : Content length : 00000059
  ORDER 20080710081658 SNIFFER : Container content:
http://www.DFH0XCMN.DFH0XCP5.Request.com{DFH0XCMNOperation}
  ORDER 20080710081658 SNIFFER : >=====
  ORDER 20080710081658 SNIFFER : Container Name : DFHWS-AUTHTOKEN
  ORDER 20080710081658 SNIFFER : Content length : 00000013
  ORDER 20080710081658 SNIFFER : Container content: UsernameToken
  ORDER 20080710081658 SNIFFER : >=====
  ORDER 20080710081658 SNIFFER : Container Name : DFHWS-TOKEN-NS
  ORDER 20080710081658 SNIFFER : Content length : 00000081
  ORDER 20080710081658 SNIFFER : Container content:
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.x
sd
  ORDER 20080710081658 SNIFFER : >=====
  ORDER 20080710081658 SNIFFER : Container Name : DFHWS-PIPELINE
  ORDER 20080710081658 SNIFFER : Content length : 00000008
  ORDER 20080710081658 SNIFFER : Container content: EXSECURE
  ORDER 20080710081658 SNIFFER : >=====
  ORDER 20080710081658 SNIFFER : Container Name : DFHWS-USERID
  ORDER 20080710081658 SNIFFER : Content length : 00000008
  ORDER 20080710081658 SNIFFER : Container content: USERWS05
  ORDER 20080710081658 SNIFFER : >=====
  ORDER 20080710081658 SNIFFER : Container Name : DFHWS-TRANID
  ORDER 20080710081658 SNIFFER : Content length : 00000004
  ORDER 20080710081658 SNIFFER : Container content: ORDS
  ORDER 20080710081658 SNIFFER : >=====
  ORDER 20080710081658 SNIFFER : Container Name :
DFHWS-WEBSERVICE
  ORDER 20080710081658 SNIFFER : Content length : 00000032
  ORDER 20080710081658 SNIFFER : Container content: placeOrder

```

In Example 10-17 on page 346 we see the contents of the containers during RECEIVE-REQUEST processing.

- ▶ **DFHREQUEST** contains the entire SOAP Request from WebSphere Application Server. Within it we can see the SOAP header containing the UsernameToken for the user ID (JOHNDOE) with the password (password).

- ▶ **DFHWS-USERID** contains PRIVAT01 which is the starting user ID from the URIMAP on corresponding to the Web service placeOrder.
- ▶ Message Handler **CIWSMSGO** executes and replaces the contents of DFHWS-TRANID with ORDS.

Message Handler **SNIFFER** executes again after the invocation of TFIM. We can now see that **DFHWS-USERID** contains USERWS05, which is the user ID passed back to CICS by the Security Token Service. And three new containers which relate to the invocation of the Security Token Service are also shown:

- ▶ **DFHWS-STS-EP** contains the endpoint of the STS server.
- ▶ **DFHWS-AUTHTOKEN** contains UsernameToken which is the token we want TFIM to issue.
- ▶ **DFHWS-TOKEN-NS** contains the namespace for valid token types.

SNIFFER Output showing Web service response

Example 10-18 contains output from the SNIFFER program showing the response processing.

Example 10-18 Output from SNIFFER during SEND-RESPONSE processing

```

ORDR 20080710082045 SNIFFER : *** Start *** SEND-RESPONSE
ORDR 20080710082045 SNIFFER : >=====
ORDR 20080710082045 SNIFFER : Container Name : DFHFUNCTION
ORDR 20080710082045 SNIFFER : Content length : 00000016
ORDR 20080710082045 SNIFFER : Container Content: SEND-RESPONSE
ORDR 20080710082045 SNIFFER : >=====
ORDR 20080710082045 SNIFFER : Container Name : DFHRESPONSE
ORDR 20080710082045 SNIFFER : Content length : 00001737
ORDR 20080710082045 SNIFFER : Container content: <SOAP-ENV:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-ENV:Body><DFHO
XCMNOperationResponse xmlns="http://www.DFHOXCMN.DFHO
XCP5.Response.com"><ca_request_id>010RDR</ca_request_id><ca_return_code>0</ca_r
eturn_code><ca_response_message>ORDER SUCCESSFULLY PLACED</ca_response_message>
<ca_order_request><ca_userid>a</ca_userid><ca_charge_dept>a</ca_charge_
dept><ca_item_ref_number>10</ca_item_ref_number><ca_quantity_req>1</ca_
quantity_req><filler1></filler1></ca_order_request></DFHOXCMNOperationR
esponse>
</SOAP-ENV:Body></SOAP-ENV:Envelope>
ORDR 20080710082045 SNIFFER : >=====
ORDR 20080710082045 SNIFFER : Container Name : DFHWS-URI
ORDR 20080710082045 SNIFFER : Content length : 00000022

```

```

ORDR 20080710082045 SNIFFER : Container content: /exampleApp/placeOrder
ORDR 20080710082045 SNIFFER : >=====
ORDR 20080710082045 SNIFFER : Container Name   : DFHWS-PIPELINE
ORDR 20080710082045 SNIFFER : Content length  : 00000008
ORDR 20080710082045 SNIFFER : Container content: EXSECURE
ORDR 20080710082045 SNIFFER : >=====
ORDR 20080710082045 SNIFFER : Container Name   : DFHWS-USERID
ORDR 20080710082045 SNIFFER : Content length  : 00000008
ORDR 20080710082045 SNIFFER : Container content: USERWS05
ORDR 20080710082045 SNIFFER : >=====
ORDR 20080710082045 SNIFFER : Container Name   : DFHWS-TRANID
ORDR 20080710082045 SNIFFER : Content length  : 00000004
ORDR 20080710082045 SNIFFER : Container content: ORDS
ORDR 20080710082045 SNIFFER : >=====
ORDR 20080710082045 SNIFFER : Container Name   :DFHWS-WEBSERVICE
ORDR 20080710082045 SNIFFER : Content length  : 00000032
ORDR 20080710082045 SNIFFER : Container content: placeOrder
ORDR 20080710082045 SNIFFER : **** End ****

```

In Example 10-18 on page 349, we see the contents of the containers during SEND-RESPONSE processing, after the call to the STS and the execution of the target CICS business logic program.

- ▶ **DFHRESPONSE** now contains the SOAP Response from CICS, which is successful.

For details on further problem determination techniques when using the CICS WS-Trust support, see Appendix B, “Problem determination” on page 369.

10.7 Using different token types

In the previous scenario, CICS receives a SOAP request message containing user credentials within a UsernameToken, and interoperates with TFIM in order to map the user ID to a RACF user ID. CICS can also interoperate with TFIM to send and receive messages that contain different types of security token, such as SAML assertions, LTPA tokens, and Kerberos tokens.

In the following scenario we show how CICS interoperates with TFIM to switch an LTPA token for a UsernameToken containing the service requester’s RACF user ID.

This scenario is shown in Figure 10-21.

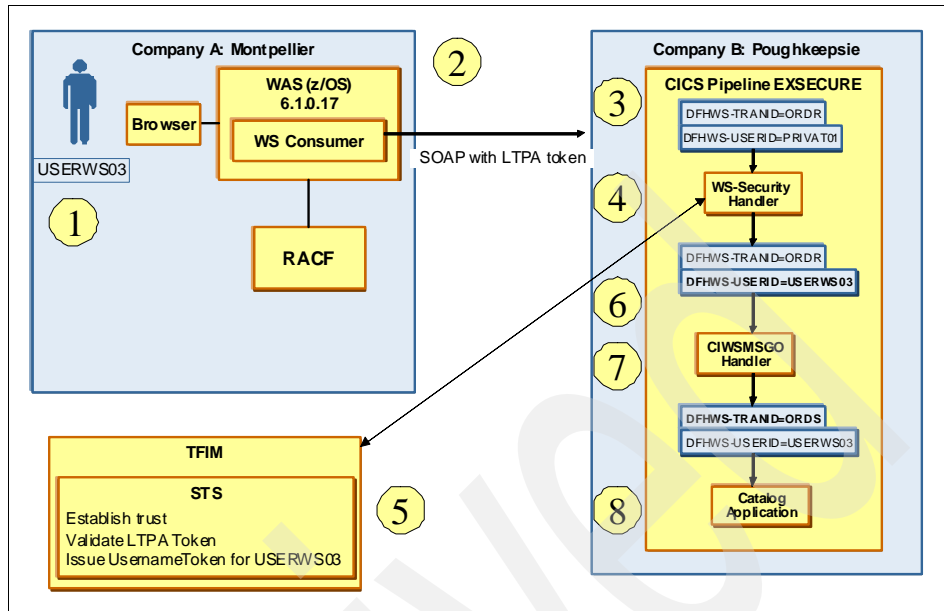


Figure 10-21 Using different token types scenario

The sequence of events is as follows:

1. An employee accesses the catalog manager Web application through a Web browser. The user authenticates with WebSphere Application Server for z/OS using basic authentication and the RACF user ID USERWS03. WebSphere validates the user's security credentials by calling RACF and creates an LTPA token.
2. When the user places an order, the J2EE application makes a Web service request to CICS. The Web service requester is configured so that the user's LTPA token is sent by WebSphere in the WS-Security SOAP header.
3. The CICS region receives the SOAP request through its TCPIP SERVICE and looks up the matching URIMAP. The URIMAP instructs CICS that requests for the path /exampleApp/placeOrder are to use the WEBSERVICE placeOrder, the PIPELINE EXSECURE, the transaction ORDR and the user ID PRIVAT01. CICS starts pipeline processing using this information.
The context containers DFHWS-USERID and DFHWS-TRANID are initialized with the current user ID, PRIVAT01, and the current tranid, ORDR.
4. The pipeline contains the CICS-supplied security handler, which is configured to use an STS. The security handler extracts the security credentials (LTPA token) from the inbound SOAP request and sends a new SOAP request, through a special internal pipeline, to the STS requesting that new credentials are issued.

5. The STS service is provided by TFIM. TFIM validates the inbound LTPA token (which contains the user ID USERWS03), and then issues a UsernameToken for USERWS03. The UsernameToken issued by TFIM does not contain a password.

Note: USERWS03 is a valid user ID for both WebSphere Application Server and CICS, TFIM therefore performs a token switching role in this scenario rather than an identity mapping.

6. The CICS-supplied security handler checks the response from TFIM and updates the context container DFHWS-USERID with the new user ID USERWS03.
7. The pipeline also includes the user-written message handler CIWSMSGO which sets the context container DFHWS-TRANID to contain ORDS for any placeOrder request.
8. As the context containers DFHWS-USERID and DFHWS-TRANID have been updated, and the terminal handler is one of the CICS-supplied SOAP handlers, the target application for placeOrder runs under user ID USERWS03 and tranid ORDS, as a new task.

10.7.1 Configuring the service requester

In this section we list the changes that we made to WebSphere Application Server for z/OS and the Web service client. We do not give step-by-step configuration instructions.

- ▶ The catalog manager Web application is configured with a security constraint so that users must authenticate and the application is executed with the caller's identity (see "Adding a Web security constraint" on page 270).
- ▶ WebSphere Application Server for z/OS is configured to use LTPA as an authentication mechanism and RACF as a user registry.
- ▶ The service requester application is configured to send an LTPA token using a procedure similar to the one described in "Configuring the request generator for identity assertion" on page 265.
- ▶ The LTPA keys are exported from WebSphere Application Server and a password is assigned (the LTPA keys are shared with TFIM when we configure TFIM to validate LTPA tokens).

10.7.2 Configuring CICS to support different token types

In this section we show how to configure CICS to accept different token types, including:

- ▶ UserNameToken
- ▶ X509 Token
- ▶ Kerberos Ticket
- ▶ SAML Assertion
- ▶ TAM Credential
- ▶ LTPA Token
- ▶ Custom Token

We do this by altering the pipeline configuration file for pipeline EXSECURE. The pipeline configuration file used in the previous WS-Trust scenario is shown in Example 10-1 on page 322.

The `<auth_token_type>` specifies what type of identity token is required. CICS selects the first security token of the specified type and then sends it to TFIM for validation. The pipeline configuration file shown in Example 10-1 on page 322. only accepts UsernameTokens.

By removing the `<auth_token_type>` element, we configure the CICS security handler to select the first security token that it finds in the `<wsse:Security>` section of the SOAP header, irrespective of the token type.

After changing the pipeline configuration file, we need to re-install the EXSECURE pipeline.

10.7.3 Configuring TFIM to validate an LTPA token

In this section we show the changes we made to our trust chain in order to support and validate an LTPA token.

We require our modified Trust Chain in TFIM to perform two tasks:

- ▶ Validate the incoming LTPA token.
- ▶ Issue a UsernameToken for the user ID contained in the LTPA token (the UsernameToken contains no password).

Note: We are not changing the user ID, just the form of the security credentials. We therefore do not need a mapping module.

To do this, we perform the following steps:

1. Expand **Tivoli Federated Identity Manager** in the left column of the Integrated Solutions Console.
2. Expand **Configure Trust Service**.
3. Select **Trust Service Chains** as shown in Figure 10-6 on page 330.
4. Select **CICS WS-Security Redbook** Trust Service Chain, and then click **Modify Chain**. The results are shown in Figure 10-22.

Modify Module Chain

Chain Identification

*Chain Name
CICS WS-Security Redbook

Description
CICS UsernameToken Chain for ITSO

Initialize the chain upon startup of the Runtime

Chain Structure

Module Instance
Select Module Instance...

Mode
Select Mode...

Add selected module instance to chain

Trust Service Chain Modules

Delete Reorder

Select	Order	Module Instance Name	Module Instance Type	Mode
<input checked="" type="radio"/>	1	Default UserNameToken	com.tivoli.am.fim.trustserver.sts.modules.UsernameTokenSTSMModule	validate
<input type="radio"/>	2	Default Map Module	com.tivoli.am.fim.trustserver.sts.modules.STSMapDefault	map
<input type="radio"/>	3	Default UserNameToken	com.tivoli.am.fim.trustserver.sts.modules.UsernameTokenSTSMModule	issue

Total: 3 Selected: 1

OK Apply Cancel

Figure 10-22 Modifying CICS WS-Security Redbook Trust Chain Part 1

5. The Default UserNameToken module with a Mode of Validate is already selected. Click **Delete**. The results can be seen in Figure 10-23.

Trust Service Chain Modules

Delete Reorder

Select	Order	Module Instance Name	Module Instance Type	Mode
<input type="radio"/>	1	Default Map Module	com.tivoli.am.fim.trustserver.sts.modules.STSMapDefault	map
<input type="radio"/>	2	Default UserNameToken	com.tivoli.am.fim.trustserver.sts.modules.UsernameTokenSTSMModule	issue

Total: 2 Selected: 0

Figure 10-23 Modifying CICS WS-Security Redbook Trust Chain Part 2

- In the **Module Instance** drop down list, select **Default LTPA Token**. In the **Mode** drop down list, select **Validate**. Then click **Add selected module instance to chain**. The results can be seen in Figure 10-24.

Trust Service Chain Modules				
		Delete Reorder		
Select	Order	Module Instance Name	Module Instance Type	Mode
<input type="radio"/>	1	Default Map Module	com.tivoli.am.fim.trustserver.sts.modules.STSMapDefault	map
<input type="radio"/>	2	Default UserNameToken	com.tivoli.am.fim.trustserver.sts.modules.UsernameTokenSTSModule	issue
<input type="radio"/>	3	Default LTPA Token	com.tivoli.am.fim.trustserver.sts.modules.STSLTPATokenModule	validate
		Total: 3 Selected: 0		

Figure 10-24 Modifying CICS WS-Security Redbook Trust Chain Part 3

- We now need to reorder the modules, so that the Default LTPA Token module for Validate executes first, and the Default UserNameToken module for Issue executes last. This is done by typing 1, 2 and 3 against the LTPA, Map and UserName modules and clicking **Reorder**.
- The incoming LTPA token contains the security credentials of a user who has already authenticated with RACF. We do not need to map the user ID so we delete the Default Map Module from the trust chain.

Select the Default Map Module and click **Delete**. The results can be seen in Figure 10-25.

Trust Service Chain Modules				
		Delete Reorder		
Select	Order	Module Instance Name	Module Instance Type	Mode
<input type="radio"/>	1	Default LTPA Token	com.tivoli.am.fim.trustserver.sts.modules.STSLTPATokenModule	validate
<input type="radio"/>	2	Default UserNameToken	com.tivoli.am.fim.trustserver.sts.modules.UsernameTokenSTSModule	issue
		Total: 2 Selected: 0		

Figure 10-25 Modifying CICS WS-Security Redbook Trust Chain Part 4

- Click **OK** to return to the Trust Service Chains screen. We next need to configure the LTPA Token module. Select the CICS WS-Security Redbook Trust Chain and then click **Properties**.

10. On the Trust Service Chain Mapping Properties screen, select the Default LTPA Token module and click **Properties**. The results can be seen in Figure 10-26.

Module Chain Item Properties

Partner properties

+Location of LTPA File
[Text Field] [Browse...]

[Import LTPA File]

+File contents of the imported LTPA file
[Text Area]

+Password for key protection (Must be the same as when keys were created.)
[Text Field]

+Password for key protection (confirm)
[Text Field]

Use the FIPS standard

[OK] [Cancel]

Figure 10-26 Modifying CICS WS-Security Redbook Trust Chain Part 5

11. Enter the location for the LTPA keys file in the first field, or click **Browse** to search for it.

Click **Import LTPA File** to import the file into TFIM, and then enter the password REDBOOKS in the two password fields (this is the password we chose when exporting the LTPA keys from WebSphere Application Server). The results are shown in Figure 10-27.

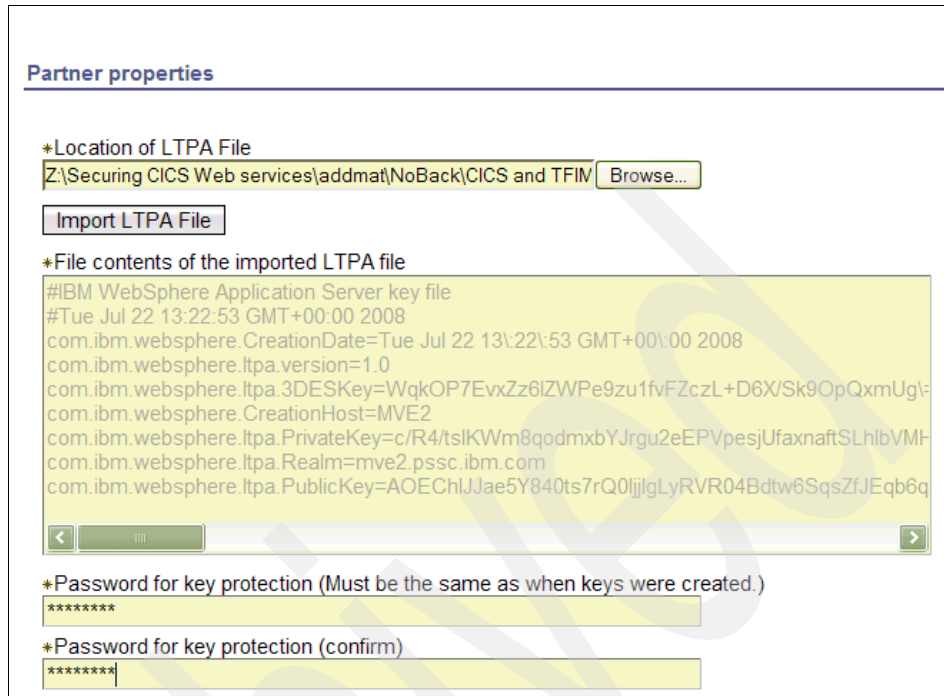


Figure 10-27 Modifying CICS WS-Security Redbook Trust Chain Part 6

12. Click **OK** to finalize the changes.

10.7.4 Testing the WS-Trust with LTPA Token scenario

In order to test our WS-Trust LTPA token scenario, we re-deploy the configured Web client application (file **ExampleAppClientV6WSTrust.ear**) to our WebSphere Application Server running on Windows. See “Installing and setting up the application” on page 174 for more information on deploying and testing the sample application.

We submit a `placeOrder` request so that a Web service request is sent from WebSphere Application Server to CICS.

In this section we show the results of testing the scenario. We show these tasks:

- ▶ Request from WebSphere to CICS
- ▶ Request from CICS to TFIM
- ▶ Response from TFIM to CICS
- ▶ CEMT showing that the tasks run under the correct user IDs

Request from WebSphere to CICS

An example of the request message with the attached LTPA token is shown in Example 10-19.

Example 10-19 SOAP Request message with LTPA Token

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Header>
<wsse:Security soapenv:mustUnderstand="1"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wss
ecurity-secext-1.0.xsd">
<wsse:BinarySecurityToken ValueType="wsst:LTPA"
xmlns:wsst="http://www.ibm.com/websphere/appserver/tokentype/5.0.2">
BS1Hf1WGE1keowZP8PjauhR1RvR0edJ1XAXQ/bUCWC8wsLT8divR5FF7v1tqF4bpg91Uc3a
bqnKypB1JL+mSmVtK09hiWZ40FCCaeyXaQpsF001eV2BFihRd3J0HyjelksL71Fk84VbeQR
qJ7VB6iRnDCKAYMpxEeVeNrZyNSfVBuHVnet171XA1CxRmezIa8EDmqDQ6QH1RYH3AZNWrm
/Ow76PVRud1A3Dr2Eio6wJtDay7A56u+i2ez3tTKR7acyJfzXF8z7wgzyRODCwI941H26MD
53NELBILxT9Jmec=
</wsse:BinarySecurityToken>
</wsse:Security>
</soapenv:Header>
<soapenv:Body>
<p635:DFHOXCMNOperation
xmlns:p635="http://www.DFHOXCMN.DFHOXCP5.Request.com">
<p635:ca_request_id>01ORDR</p635:ca_request_id>
<p635:ca_return_code>0</p635:ca_return_code>
<p635:ca_response_message>^C030183018</p635:ca_response_message>
<p635:ca_order_request>
<p635:ca_user ID>NIGEL</p635:ca_user ID>
<p635:ca_charge_dept>PSSC</p635:ca_charge_dept>
<p635:ca_item_ref_number>10</p635:ca_item_ref_number>
<p635:ca_quantity_req>1</p635:ca_quantity_req>
<p635:filler1 xsi:nil="true"/>
</p635:ca_order_request>
</p635:DFHOXCMNOperation>
</soapenv:Body>
</soapenv:Envelope>
```

- ▶ The <wsse:Security> header includes the mustUnderstand="1" attribute, which indicates that either this header must be processed or a SOAP fault thrown.
- ▶ The <wsse:BinarySecurityToken> element has a ValueType="wsst:LTPA" attribute, and a namespace declaration of xmlns:wsst="http://www.ibm.com/websphere/appserver/tokentype/5.0.2". Together these indicate that this is an LTPA Version 1 token.

Request from CICS to TFIM

The WS-Trust handler within the pipeline makes a RequestSecurityToken request to TFIM. An example of this message is shown in Example 10-20, with the SOAP Envelope and Body tags removed.

Example 10-20 RequestSecurityToken request sent to TFIM

```

<wst:RequestSecurityToken
xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wss-
ecurity-secext-1.0.xsd">
<wst:RequestType>http://schemas.xmlsoap.org/ws/2005/02/trust/Issue</wst:
RequestType>
<wst:TokenType>http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd/UsernameToken</wst:TokenType>
  <wst:Base>
    <wsse:BinarySecurityToken
xmlns:wsst="http://www.ibm.com/websphere/appserver/tokentype/5.0.2"
ValueType="wsst:LTPA">BS1Hf1WGE1keowZP8PjauhR1RvR0edJ1XAXQ/bUCWC8wsLT8d
ivR5FF7v1tqF4bpg91Uc3abqnKYpB1JL+mSmVtK09hiWZ40FCCaeyXaQpsF001eV2BFIhRd
3J0Hyje1ksL71Fk84VbeQRqJ7VB6iRnDCKAYMpxEeVeNrZyNSfVBuHVnet171XA1CxRmezI
a8EDmqDQ6QH1RYH3AZNWrm/OW76PVRud1A3Dr2Eio6wJtDay7A56u+i2ez3tTKR7acyJfzX
F8z7wgzYRODCwI941H26MD53NELBILxT9Jmec=</wsse:BinarySecurityToken>
    </wst:Base>
    <wsp:AppliesTo>
      <wsa:EndpointReference>
<wsa:Address>/exampleApp/placeOrder/DFHOXCMN0peration</wsa:Address>
      </wsa:EndpointReference>
    </wsp:AppliesTo>
  </wst:RequestSecurityToken>

```

- ▶ The SOAP body contains the <wst:RequestSecurityToken> element which contains a number of child elements:
 - <wst:RequestType> contains the type of the request from the WS-Trust specification. This is an issue request.
 - <wst:TokenType> contains the type of token that CICS expects TFIM to return. For CICS as a Web service provider, this token is always a UsernameToken. For CICS as a Web service requester, the token type is specified in <auth_token_type> in the pipeline config file.
 - <wst:Base> contains the LTPA token that CICS has extracted from the incoming message. We can see the LTPA token within the <BinarySecurityToken> element.
 - <wsp:AppliesTo> contains a child element <wsa:EndpointReference> which has a child element <wsa:Address>. This contains the URI of the Web service that the user is attempting to access.

Response from TFIM to CICS

TFIM processes the RequestSecurityToken and sends back a RequestSecurityTokenResponse. An example of this message is shown in Example 10-21, with the SOAP Envelope and Body tags removed.

Example 10-21 RequestSecurityTokenResponse received from TFIM

```

<wst:RequestSecurityTokenResponse Context=""
wsu:Id="uuid550a5057-011b-126b-890d-da369c191407"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
curity-utility-1.0.xsd">
<wst:TokenType>http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd/UsernameToken</wst:TokenType>
  <wsp:AppliesTo
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <wsa:EndpointReference>

<wsa:Address>/exampleApp/placeOrder/DFHOXCMNOperation</wsa:Address>
    </wsa:EndpointReference>
  </wsp:AppliesTo>
  <wst:RequestedSecurityToken>

```



```

    <wss:UsernameToken
wsu:Id="username550a5c7b-011b-1cb4-ac3f-da369c191407"
xmlns:wss="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
curity-secext-1.0.xsd"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
curity-utility-1.0.xsd">
    <wss:Username>USERWS03</wss:Username>
    <wss:Nonce
EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-s
oap-message-security-1.0#Base64Binary">5NdokYNZUIxbT0kab5DKJQ==</wss:No
nce>
    <wsu:Created>2008-07-24T12:28:07Z</wsu:Created>
    </wss:UsernameToken>
  </wst:RequestedSecurityToken>
  <wst:RequestedAttachedReference
xmlns:wss="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
curity-secext-1.0.xsd">
    <wss:SecurityTokenReference>
    <wss:Reference
URI="#username550a5c7b-011b-1cb4-ac3f-da369c191407"
ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-user
name-token-profile-1.0#UsernameToken"
xmlns:wss="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
curity-secext-1.0.xsd"/>
    </wss:SecurityTokenReference>
  </wst:RequestedAttachedReference>
</wst:RequestSecurityTokenResponse>

```

-
- ▶ The SOAP Body contains the <wst:RequestSecurityTokenResponse> element which contains two child elements, <wst:RequestedSecurityToken> and <wst:RequestedAttachedReference>.
 - <wst:RequestedSecurityToken> contains the token returned from TFIM.
 - <wss:UsernameToken> is the immediate child element of <wst:RequestedSecurityToken> and contains three child elements:
 - <wss:Username> contains the user ID **USERWS03** extracted from the LTPA token
 - <wss:Nonce> is an element tolerated by CICS but not used.
 - <wss:Created> is an element tolerated by CICS but not used.
 - <wst:RequestedAttachedReference> contains the schema definitions of the UsernameToken.

CEMT INQUIRE TASK

Example 10-22 shows the ORDS transaction running with the user ID USERWS03 (the user ID returned from TFIM), while the ORDR transaction continues to execute with the PRIVAT01 user ID.

Example 10-22 ORDS executing with user ID returned from TFIM

INQUIRE TASK

STATUS: RESULTS - OVERTYPE TO MODIFY

Tas(0000075) Tra(**ORDR**) Sus Tas Pri(001)

Sta(U) Use(**PRIVAT01**) Uow(C2ABAAF9247F2D04) Hty(RZCBNOTI)

Tas(0000076) Tra(**ORDS**) Sus Tas Pri(001)

Sta(U) Use(**USERWS03**) Uow(C2ABAAF95D2F7B4B) Hty(EDF)

XSLT example

Example A-1 is a listing of the XSLT file `xtractCertificateFromSignature.xsl`, which is used for stripping an XML digital signature in “Stripping the XML digital signature” on page 302.

Example: A-1 ExtractCertificateFromSignature.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:wsse10="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-w
ssecurity-secext-1.0.xsd"
xmlns:wsse-d12="http://schemas.xmlsoap.org/ws/2002/07/secext"
xmlns:wsse-d13="http://schemas.xmlsoap.org/ws/2003/06/secext"
xmlns:wsu10="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-ws
security-utility-1.0.xsd"
xmlns:wsu-d12="http://schemas.xmlsoap.org/ws/2002/07/utility"
xmlns:wsu-d13="http://schemas.xmlsoap.org/ws/2003/06/utility"
xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:dsig="http://www.w3.org/2000/09/xmldsig#"
exclude-result-prefixes="wsse10 wsse-d12 wsse-d13 wsu10 wsu-d12
wsu-d13 SOAP dsig"
version="1.0">

<xsl:output method="xml" />
```

```

(1)<xsl:template name="strip-signature-1.0">
  <xsl:param name="header" />
  <xsl:variable name="stripped-header-strip-signature">
    <wsse10:Security>
      <xsl:for-each select="$header/*">
        <xsl:choose>
          <xsl:when
test="namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200
401-wss-wssecurity-secext-1.0.xsd' and
//wsse10:Security/dsig:Signature/dsig:KeyInfo/wsse10:SecurityTokenRefer
ence/wsse10:Reference/@URI = concat('#',@wsu10:Id)">
          <xsl:copy-of select="." />
          <xsl:message>Keep the BST</xsl:message>
        </xsl:when>
        <xsl:when
test="namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and
local-name()='Signature'">
          <xsl:message>Remove the SIG</xsl:message>
        </xsl:when>
        <xsl:otherwise>
          <xsl:copy-of select="." />
        </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each>
    </wsse10:Security>
  </xsl:variable>
  <xsl:copy-of select="$stripped-header-strip-signature" />
</xsl:template>

(2)<xsl:template name="strip-signature-draft-12">
  <xsl:param name="header" />
  <xsl:variable name="stripped-header">
    <wsse-d12:Security>
      <xsl:for-each select="$header/*">
        <xsl:choose>
          <xsl:when
test="namespace-uri()='http://schemas.xmlsoap.org/ws/2002/07/secext'
and
//wsse-d12:Security/dsig:Signature/dsig:KeyInfo/wsse-d12:SecurityTokenR
eference/wsse-d12:Reference/@URI = concat('#',@wsu-d12:Id)">
          <xsl:copy-of select="." />
          <xsl:message>Keep the BST</xsl:message>
        </xsl:when>
        <xsl:when

```

```

        <xsl:when
test="namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and
local-name()='Signature' ">
            </xsl:when>
            <xsl:otherwise>
                <xsl:copy-of select="." />
            </xsl:otherwise>
        </xsl:choose>
    </xsl:for-each>
</wsse-d12:Security>
</xsl:variable>
<xsl:copy-of select="$stripped-header" />
</xsl:template>

(3)<xsl:template name="strip-signature-draft-13">
    <xsl:param name="header" />
    <xsl:variable name="stripped-header">
        <wsse-d13:Security>
            <xsl:for-each select="$header/*">
                <xsl:choose>
                    <xsl:when
test="namespace-uri()='http://schemas.xmlsoap.org/ws/2003/06/secext'
and
//wsse-d13:Security/dsig:Signature/dsig:KeyInfo/wsse-d13:SecurityTokenR
eference/wsse-d13:Reference/@URI = concat('#',@wsu-d13:Id)">
                        </xsl:when>
                    <xsl:when
test="namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and
local-name()='Signature' ">
                            </xsl:when>
                            <xsl:otherwise>
                                <xsl:copy-of select="." />
                            </xsl:otherwise>
                        </xsl:choose>
                    </xsl:for-each>
                </wsse-d13:Security>
            </xsl:variable>
            <xsl:copy-of select="$stripped-header" />
        </xsl:template>

(4)<xsl:template match="wsse10:Security">
    <xsl:variable name="stripped-header">

```

```

    <xsl:call-template name="strip-signature-1.0">
      <xsl:with-param name="header" select="." />
    </xsl:call-template>
  </xsl:variable>
  <xsl:if test="$stripped-header/wsse10:Security/*">
    <xsl:copy>
      <xsl:copy-of select="@*" />
      <xsl:copy-of
        select="$stripped-header/wsse10:Security/*" />
    </xsl:copy>
  </xsl:if>
</xsl:template>

```

```

(5)<xsl:template match="wsse-d12:Security">
  <xsl:variable name="stripped-header">
    <xsl:call-template name="strip-signature-draft-12">
      <xsl:with-param name="header" select="." />
    </xsl:call-template>
  </xsl:variable>
  <xsl:if test="$stripped-header/wsse-d12:Security/*">
    <xsl:copy>
      <xsl:copy-of select="@*" />
      <xsl:copy-of
        select="$stripped-header/wsse-d12:Security/*" />
    </xsl:copy>
  </xsl:if>
</xsl:template>

```

```

(6)<xsl:template match="wsse-d13:Security">
  <xsl:variable name="stripped-header">
    <xsl:call-template name="strip-signature-draft-13">
      <xsl:with-param name="header" select="." />
    </xsl:call-template>
  </xsl:variable>
  <xsl:if test="$stripped-header/wsse-d13:Security/*">
    <xsl:copy>
      <xsl:copy-of select="@*" />
      <xsl:copy-of
        select="$stripped-header/wsse-d13:Security/*" />
    </xsl:copy>
  </xsl:if>
</xsl:template>

```

```

(7)<xsl:template match="@*">
  <xsl:copy />

```

```
</xsl:template>

(8)<xsl:template match="*">
  <xsl:copy>
    <xsl:copy-of select="@*" />
    <xsl:apply-templates select="node()" />
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

The `<xsl:stylesheet ...>` element is used to define the definitions of the different namespaces that are used in this XSLT:

- ▶ The namespaces with prefix **wsse** relate to the Web Services Security Extension.
- ▶ The namespaces with prefix **wsu** relate to the Web Services Utility extensions.
- ▶ Finally, we have the namespace definitions for SOAP, and the specification for XML Digital Signatures.

The `<xsl:output method="xml" />` element specifies that we are going to create an XML file (as opposed to a text file, for example).

The elements (1), (2) and (3) shown in Example A-1 (that start with `<xsl:template name=`) remove the Signature element.

The elements (4), (5) and (6) shown in Example A-1 (that start with `<xsl:template match=`) match the Security element for the different possible wsse namespaces and call the elements (1), (2) or (3) in order to remove the Signature element.

The elements (7) and (8) shown in Example A-1 copy all the elements and attributes of the message (other than the signature element).

Archived

Problem determination

In this appendix, we discuss what errors you might see when running your CICS system with a Security Token Server, and we explain several techniques that you can use to diagnose these problems.

We start with our Web services client receiving a SOAP Fault from CICS (see Example B-1). In compliance with the WS-Security specification, CICS does not return full and exact details as to what was wrong with the security credentials in order to prevent possible security attacks. Accordingly, the SOAP Fault below merely indicates that a successful response from the STS was not received.

Example: B-1 SOAP Fault received due to security token error

```
<SOAP-ENV:Fault>  
<faultcode>wsse:FailedAuthentication</faultcode>  
<faultstring>A security token could not be authenticated</faultstring>  
</SOAP-ENV:Fault>
```

This fault could be seen for any of the following reasons:

- ▶ The security token is invalid (not recognized, unknown user ID, incorrect password)
- ▶ TFIM is unreachable (not started, network problems)
- ▶ CICS times out waiting for a response from TFIM

For more details, it is necessary to look at the CICS diagnostics, in particular, CICS messages and trace entries.

CICS messages and codes

DFHPI0512 messages indicate that the CICS Pipeline Manager has received a fault from a Secure Token Service, and then give more details about the type of the fault. These messages can be found in the MSGUSR dataset.

Later in this section, we work through a troubleshooting method showing how you could diagnose the root cause of the error message seen in Example B-2.

Example: B-2 DFHPI0512 message showing Authentication Failure with STS

DFHPI0512 07/11/2008 03:05:49 A6POS3C5 ORDR The CICS Pipeline Manager has received a fault from the target Secure Token Service:

http://mett.pdl.pok.ibm.com:9610/TrustServer/SecurityTokenService. The fault had a fault code of p383:FailedAuthentication.

Other error messages you might see are shown in Example B-3.

Example: B-3 DFHPI messages displayed when TFIM was not started

DFHPI0400 07/10/2008 04:22:22 A6POS3C5 ORDR The CICS pipeline HTTP transport mechanism failed to send a request because there was a socket error.

DFHPI0504 07/10/2008 04:22:22 A6POS3C5 ORDR The CICS Pipeline Manager has failed to communicate with a remote server due to an error in the underlying transport. TRANSPORT: HTTP, PIPELINE: DFHPITCR.

DFHPI0512 07/10/2008 04:22:22 A6POS3C5 ORDR The CICS Pipeline Manager has received a fault from the target Secure Token Service:

http://mett.pdl.pok.ibm.com:9610/TrustServer/SecurityTokenService. The fault had a fault code of SOAP-ENV:Server.

Example B-3 shows the error we received after we shut down the TFIM server and then attempted to use it. The Pipeline Manager invokes the Web Domain's Web Client, which in turn uses the Sockets Domain to open a socket to TFIM. The socket connect fails because TFIM is not started.

Trace Point SO 0202 is displayed in Example B-4, Trace Point WB 0709 in Example B-5, and Trace Point PI 0A32 in Example B-6. This shows that CICS

returns an Internal Server Error SOAP Fault to the Trust Client in Example B-7, which then builds the standard SOAP Fault as seen before in Example B-1 on page 369.

Example: B-4 SO=1 SO 0202 trace; cannot connect to TFIM as it was stopped

```
SO 0202 SOCK EXIT - FUNCTION(CONNECT) RESPONSE(EXCEPTION)
REASON(NO_CONNECTION)
```

Example: B-5 WB=1 WB 0701 trace; socket error opening session

```
WB 0701 WBCL EXIT - FUNCTION(OPEN_SESSION) RESPONSE(EXCEPTION)
REASON(SOCKET_ERROR)
```

Example: B-6 PI=2 PI 0A32; CICS has built a SOAP fault to encapsulate the error

```
PI 0A32 PIIS EVENT - RESPONSE_CNT -
```

```
TASK-02730 KE_NUM-00ED TCB-L8000/007B74F8 RET-A6043EAO
TIME-04:26:14.2404899389 INTERVAL-00.0000003281          =002550=
```

```
1-502D454E *<SOAP-ENV:Envelope xmlns:SOAP-EN*
672F736F *V="http://schemas.xmlsoap.org/so*
3E3C534F *ap/envelope/"><SOAP-ENV:Body><S0*
636F6465 *AP-ENV:Fault xmlns=""><faultcode*
3C666175 *>SOAP-ENV:Server</faultcode><fau*
6F723C2F *ltstring>Internal Server Error</*
3E3C2F53 *faultstring></SOAP-ENV:Fault></S*
6C6F7065 *OAP-ENV:Body></SOAP-ENV:Envelope*
3E *> *
```

Example: B-7 SOAP Fault returned to Client (compare with Example B-6)

```
PI 0A32 PIIS EVENT - RESPONSE_CNT -
```

```
TASK-02730 KE_NUM-00ED TCB-L8000/007B74F8 RET-A684E998
TIME-04:26:14.2472810498 INTERVAL-00.0000003281          =002727=
```

```
1-502D454E *<SOAP-ENV:Envelope xmlns:SOAP-EN*
672F736F *V="http://schemas.xmlsoap.org/so*
703A2F2F *ap/envelope/" xmlns:wst="http://*
30322F74 *schemas.xmlsoap.org/ws/2005/02/t*
732E6F61 *rust" xmlns:wsse="http://docs.oa*
69732D32 *sis-open.org/wss/2004/01/oasis-2*
312E302E *00401-wss-wssecurity-secext-1.0.*
563A4661 *xsd"><SOAP-ENV:Body><SOAP-ENV:Fa*
```

```
7468656E *ult<<faultcode>wsse:FailedAuthen*
72696E67 *tication</faultcode><faultstring*
62652061 *>A security token could not be a*
534F4150 *uthenticated</faultstring></SOAP*
2F534F41 *-ENV:Fault></SOAP-ENV:Body></SOA*
*P-ENV:Envelope> *
```

As the request to the Secure Token Service is a Web Services Request flowing out from CICS, it is possible that the request might time out before completion. For example, if the TFIM providing the STS is behind several firewalls, running full diagnostic trace, the network and CPUs are busy, then the messages in Example B-8 might be seen.

Example: B-8 Request to the STS has timed out

```
DFHPI0403 07/10/2008 04:16:27 A6POS3C5 ORDR The CICS pipeline HTTP
transport mechanism failed to receive a response because the
socket receive was timed out.
DFHPI0997 07/10/2008 04:16:27 A6POS3C5 ORDR DFHPITCR The CICS pipeline
manager has encountered an error: timeout occurred.
```

The User response for message DFHPI0403 suggests considering changing the RESPWAIT parameter on the Requester Pipeline. This is not possible for STS requests because even though they flow along a pipeline, it is a special internal pipeline called DFHPITCR that cannot be modified. Therefore the default timeout values apply, which are 10 seconds for HTTP/S and 60 seconds for JMS/MQ.

CICS Trace of the Pipeline Manager Domain

In this section we show what can be seen with Pipeline Level 1 and Level 2 Trace. We first show what can be seen with PI=1 Trace for the working scenario. We then look at PI=1 and Exception trace for the failure scenario above, and look at what additional information we can obtain from PI=2 tracing.

With PI=1 tracing active, it is possible to see the activities undertaken by CICS in order to invoke TFIM as its Security Token Service. Trace entry 1700 in domain PI followed by the characters PITC can be used to identify the entry into the CICS internal Trust Client interface (see Example B-9), and trace entry 1701 followed by the characters PITC can be used to identify the exit (see Example B-10).

Example: B-9 Entry to the CICS Trust Client

```
PI 1700 PITC ENTRY - FUNCTION(TRUST_CLIENT) CHANNEL_TOKEN(28004030)
POOL_TOKEN(272F4060) WSSE_PROGRAM(DFHWSSE1) WSSE_CONFIG(2801B140
, 00000A11) MODE(PROVIDER) DIRECTION(FORWARD)
```

Example: B-10 Exit from the CICS Trust Client successfully

```
PI 1701 PITC EXIT - FUNCTION(TRUST_CLIENT) RESPONSE(OK)
```

At the same Trace Point can be found details of the operation requested of TFIM. In Example B-11, we can see that our pipeline has sent a request of type “Issue” to TFIM.

Example: B-11 PI 1700 PITC ENTRY - FUNCTION(ISSUE)

```
PI 1700 PITC ENTRY - FUNCTION(ISSUE) DESTINATION_URI_BLOCK(28105948 ,
00000041) SERVICE_URI_BLOCK(281058D8 , 00000016)
SECURITY_TOKEN_BLOCK(2810538A , 000000DD)
AUTHTOKEN_TYPE_BLOCK(26048BC2 , 0000005F) RETURNED_SECTOK_BUFF(270DE9CC
, 00000000 , 00000100) POOL_TOKEN(272F4060)
OPERATION_BLOCK(270DE9CC , 00000011)
```

All the details of the request can be seen in the trace. Parameter 3 in Example B-12 shows the Secure Token Service’s endpoint, which is fully qualified. For this scenario we are using a Tivoli Federated Identity Manager provided by the Poughkeepsie Development Laboratory.

Example: B-12 Parameter 3 of Function Issue, the STS Endpoint

```
3-39363130 *http://mett.pdl.pok.ibm.com:9610*
72766963 */TrustServer/SecurityTokenServic*
65 *e *
```

The AppliesTo value can be seen as parameter 4 in Example B-13. It can be seen that the AppliesTo value is the same as the path in the URIMAP, /exampleApp/placeOrder.

Example: B-13 Parameter 4, the AppliesTo value

```
4-6572 */exampleApp/placeOrder *
```

The security token sent to TFIM can be seen as parameter 5 in Example B-14. Our Security Token Service will validate this username token against WebSphere Registry as the first stage of its processing.

Example: B-14 Parameter 5, the Security Token (in this example a UsernameToken)

```
5- 65726E61 *<wsse:UsernameToken><wsse:Userna*
7373653A *me>JOHNDOE</wsse:Username><wsse:*
61736973 *Password Type="http://docs.oasis*
32303034 *-open.org/wss/2004/01/oasis-2004*
```

```

652D312E *01-wss-username-token-profile-1.*
73653A50 *0#PasswordText">password</wsse:P*
3E      *assword></wsse:UsernameToken> *

```

Finally, we can see the token type passed to TFIM by CICS. The pipeline configuration file for EXSECURE is configured only to accept UsernameTokens as valid security tokens. It is possible for any security type to be accepted as potentially valid and sent to TFIM, and in that case this parameter would allow you to check what type was used.

Example: B-15 Parameter 6, the Token Type (a UsernameToken)

```

6-73732F32 *http://docs.oasis-open.org/wss/2*
63757269 *004/01/oasis-200401-wss-wssecuri*
6B656E *ty-secext-1.0.xsd/UsernameToken *

```

As part of problem diagnostics, we sent in a request to /placeOrder but this time having made some changes. We saw message DFHP10512, which is the message indicating that “CICS Pipeline Manager has received a fault from the target Secure Token Service”. Running the Trace Formatter against the Auxiliary Trace dataset which was captured with PI=1 is shown in Example B-16.

Example: B-16 Exit from the CICS Trust Client with an exception

```

PI 1701 PITC EXIT - FUNCTION(TRUST_CLIENT) RESPONSE(EXCEPTION)
REASON(TRUST_FAULT)

```

We now know that there was an authentication error (from MSGUSR) and a Trust Fault (from PI=1 trace).

Pipeline Exception Trace Point PI 1706 is used to record a request failure from the Trust Client. The formatted trace of this entry can be seen in Example B-17. The fault code and fault string from the STS are displayed in ASCII format, which, as can be seen here, is not readily readable on an EBCDIC system.

Example: B-17 PI 1706 Trace Point: WST_REQUEST_FAILED

```

PI 1706 PITC *EXC* - WST_REQUEST_FAILED -

TASK-00124 KE_NUM-00EF TCB-L8000/007B74F8 RET-A60463C8
TIME-03:08:01.3074725483 INTERVAL-00.0000008906          =002572=

1-0000 2F2F7363 *../.%..?...._%>.....*
0020 2F747275 *.._/...%?.?/..?.....*
0040 6E3C2F66 *...../.%.....>.../..?>...*
0060 31383145 */.%..?...../.%.....>.....*

```

```

0080 63617469 *...../....>.../..*
00A0 2F666175 *?>..?.....$|.+.|.../..%...../.*
00C0 2F3E      *%.....>...../..%./..?....      *

```

PI=2 level tracing

If PI=2 tracing is active, Trace Points PI 0A31 and PI 0A32 will be captured, and in the formatted trace will show the contents of any request and response containers. In the processing of a pipeline, whether provider or requester, that contains a call to an STS, the first instance of a response container should be that of the response from the Secure Token Service. It is easier to look for the response first, and then search backwards (F 'PI 0A31' PREV) for the Request Container, because there can be many references to the inbound/outbound SOAP Request to/from CICS, and only one to the request to the STS.

PI 0A31 and PI 0A32 also format the ASCII data in readable format. See Example B-18.

Example: B-18 PI 0A31 Trace showing the request made to TFIM

PI 0A31 PIIS EVENT - REQUEST_CNT -

```

TASK-00128 KE_NUM-00ED TCB-L8000/007B74F8 RET-A6043EA0
TIME-04:44:46.3042165339 INTERVAL-00.0000003125          =002523=
.....Lines Deleted.....
75657374 *e/" ><SOAP-ENV:Body><wst:Request*
703A2F2F *SecurityToken xmlns:wst="http://*
30322F74 *schemas.xmlsoap.org/ws/2005/02/t*
6D61732E *rust" xmlns:wsa="http://schemas.*
7373696E *xmlsoap.org/ws/2004/08/addressin*
2E786D6C *g" xmlns:wsp="http://schemas.xml*
786D6C6E *soap.org/ws/2004/09/policy" xmln*
656E2E6F *s:wsse="http://docs.oasis-open.o*
7773732D *rg/wss/2004/01/oasis-200401-wss-*
7773743A *wssecurity-secext-1.0.xsd"><wst:*
6D6C736F *RequestType>http://schemas.xmlso*
653C2F77 *ap.org/ws/2005/02/trust/Issue</w*
653E6874 *st:RequestType><wst:TokenType>ht*
2F323030 *tp://docs.oasis-open.org/wss/200*
72697479 *4/01/oasis-200401-wss-wssecurity*
6E3C2F77 *-secext-1.0.xsd/UsernameToken</w*
3A557365 *st:TokenType><wst:Base><wsse:Use*
203C7773 *rnameToken>.. <ws*
65726E61 *se:Username>JOHNDOE</wsse:Userna*
61737377 *me>.. <wsse:Passw*
6F70656E *ord Type="http://docs.oasis-open*

```

```

312D7773 *.org/wss/2004/01/oasis-200401-ws*
23506173 *s-username-token-profile-1.0#Pas*
7373776F *swordText">fakeword</wsse:Passwo*
6420786D *rd>.. <wsu:Created xm*
70656E2E *lns:wsu="http://docs.oasis-open.*
2D777373 *org/wss/2004/01/oasis-200401-wss*
3E323030 *-wssecurity-utility-1.0.xsd">200*
43726561 *8-07-11T08:44:46+00:00</wsu:Crea*
73743A42 *ted></wsse:UsernameToken></wst:B*
6F696E74 *ase><wsp:AppliesTo><wsa:Endpoint*
706C6541 *Reference><wsa:Address>/exampleA*
696F6E3C *pp/placeOrder/DFHOXCMNOperation<*
65666572 */wsa:Address></wsa:EndpointRefer*
65717565 *ence></wsp:AppliesTo></wst:Reque*
6F64793E *stSecurityToken></SOAP-ENV:Body>*
* </SOAP-ENV:Envelope> *

```

In Example B-18 on page 375, RequestSecurityToken request, John Doe's user ID JOHND0E can be clearly seen, as can the password which will be passed to TFIM for validation (Example B-19).

Example: B-19 Trace Point PI 0A32 showing a fault from STS

PI 0A32 PIIS EVENT - RESPONSE_CNT -

```

TASK-00128 KE_NUM-00ED TCB-L8000/007B74F8 RET-A6043EAO
TIME-04:44:46.3050240026 INTERVAL-00.0000004375 =002530=
.....Lines Deleted.....
3A736F61 *nv:Body><soapenv:Fault xmlns:soa*
2E6F7267 *penv="http://schemas.xmlsoap.org*
786D6C6E */soap/envelope/"><faultcode xmln*
61702E6F *s:p383="http://schemas.xmlsoap.o*
696C6564 *rg/ws/2005/02/trust">p383:Failed*
61756C74 *Authentication</faultcode><fault*
52656769 *string>FBTSTS181E WebSphere Regi*
204A4F48 *stry authentication for user JOH*
61756C74 *NDOE failed.</faultstring><fault*
70656E76 *actor/></soapenv:Fault></soapenv*
3E *:Body></soapenv:Envelope> *

```

Example B-19 shows the response from our STS, which is a SOAP Fault. The further information provided by the faultstring informs us that WebSphere Registry authentication for JOHND0E has failed.

From the logs of our TFIM server, we can see the error returned from WebSphere Registry, and also a PasswordCheckFailedException, see Example B-20. With this information, we now check the password used by our application and discover that it had been changed to fakeword in the EAR file. This is then changed back to password and the application runs to completion successfully.

Example: B-20 TFIM error messages indicating the nature of our problem

```
Message: com.ibm.websphere.security.PasswordCheckFailedException
    at
    com.ibm.ws.security.z0S.PlatformCredentialManager.createPasswordCredential(PlatformCredentialManager.java:182)
    at
    com.ibm.ws.security.z0S.PlatformCredentialManager.createPasswordCredential(PlatformCredentialManager.java:128)
    at
    com.ibm.ws.security.registry.z0S.SAFRegistryImpl.checkPassword(SAFRegistryImpl.java:278)
    at
    com.ibm.ws.security.registry.UserRegistryImpl.checkPassword(UserRegistryImpl.java:309)
    ...Lines deleted
```

```
Trace: 2008/07/1109:40:25.68101t=9C0730c=UNKkey=P8 (13007002)
    ThreadId: 00000022
    FunctionName: validatePasswordUsingWASRegistry(String, String, UsernameTokenConfiguration)
    SourceId:
    com.tivoli.am.fim.trustserver.sts.modules.UsernameTokenSTSModule
    Category: INFO
    ExtendedMessage:
    com.tivoli.am.fim.trustserver.sts.STSModuleException: FBTSTS181E
    WebSphere Registry authentication for user JOHND
    OE failed.
    at
    com.tivoli.am.fim.trustserver.sts.modules.UsernameTokenSTSModule.validatePasswordUsingWASRegistry(UsernameTokenSTSModule.java:573)
    at
    com.tivoli.am.fim.trustserver.sts.modules.UsernameTokenSTSModule.validatePassword(UsernameTokenSTSModule.java:460)
    Caused by: javax.xml.rpc.soap.SOAPFaultException: FBTSTS181E WebSphere
    Registry authentication for user JOHND OE failed.
```

```
    at
com.tivoli.am.fim.trustserver.service.SecurityTokenService.makeFaultFrom
Exception(SecurityTokenService.java:774)
    at
com.tivoli.am.fim.trustserver.service.SecurityTokenService.processReque
st(SecurityTokenService.java:565)
    at
com.tivoli.am.fim.trustserver.service.SecurityTokenService.processReque
stSecurityToken(SecurityTokenService.java:502)
    at
com.tivoli.am.fim.trustserver.service.SecurityTokenService.requestSecur
ityToken(SecurityTokenService.java:353)
```

Sample message handler

The CIWSMSGO message handler program, listed here in Example C-1, is used to switch the transaction ID from ORDR to ORDS for placeOrder Web service requests.

Example: C-1 CIWSMSGO message handler program

```

000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID. CIWSMSGO
000300*****
000400*
000500* This program:-
000600* 1. Sets the tranid to ORDS if placeOrder
000800*****
000900
001000 AUTHOR.
001100 DATE-COMPILED.
001200 ENVIRONMENT DIVISION.
001300 CONFIGURATION SECTION.
001400 SPECIAL-NAMES.
001500 DATA DIVISION.
001600 WORKING-STORAGE SECTION.
001700 01 WS-START.
001800* Nesting offset for DISPLAYS
001900 03 NN PIC X(11) VALUE 'CIWSMSGO: '.
002000

```

```

002100 03 RESP PIC S9(8) COMP-5 SYNC.
002200 03 RESP2 PIC S9(8) COMP-5 SYNC.
002300 03 BOD-PTR USAGE IS POINTER.
002400 03 BOD-LEN PIC S9(8) COMP-4.
002500 03 CONTAINER-LEN PIC S9(8) BINARY.
002600 03 GETMAIN-PTR USAGE IS POINTER.
002700 03 GETMAIN-LEN PIC S9(8) COMP-4.
002800 03 WS-FAULT-STRING PIC X(21) value spaces.
002900 03 WS-DFHFUNCTION PIC X(16) value spaces.
003000 03 WS-WEBSERVICES PIC X(30) value spaces.
003100 03 WS-WEBSERVICES-LEN PIC S9(8) BINARY.
003200*
003300* Not Found SOAP Fault Detail section
003400*
003500 01 WS-Fault-NotFnd.
003600 03 WS-Fault-Namespace pic x(53).
003700 03 WS-Fault-RC-Lit pic X(17).
003800 03 WS-Fault-RC pic X(2).
003900 03 WS-Fault-RC-ELit pic X(18).
004000 03 WS-Fault-Item-Lit pic X(11).
004100 03 WS-Fault-Item pic X(4).
004200 03 WS-Fault-Item-ELit pic X(12).
004300 03 WS-Fault-ENamespace pic x(19).
004400
004500 01 EXPARSER-COMLEN PIC S9(4) COMP-4.
004600 01 CA-PARSER-RSP.
004700 03 CA-PARSER-REQUEST.
004800 05 FILLER PIC X(2).
004900 05 CA-TRANID PIC X(4).
005000 03 CA-PARSER-REF-REQ pic x(4).
005100 03 CA-PARSER-RET-CODE pic x(2).
005200 03 FILLER PIC X(200).
005300
005400*****
005500* Externally referenced data areas
005600*****
*
005700 LINKAGE SECTION.
005800 01 BOD-AREA.
005900 02 FILLER PIC X OCCURS 64000 DEPENDING ON BOD-LEN.
006000
006100 01 GETMAIN-AREA.
006200 02 FILLER PIC X OCCURS 64000 DEPENDING ON GETMAIN-LEN.
006300 01 CONTAINER-DATA.
006400 05 FILLER PIC X OCCURS 32768

```

```

006500                                DEPENDING ON CONTAINER-LEN.
006600*-----*
006700    EJECT
006800*-----*
006900    PROCEDURE DIVISION.
007000*-----*
007100        EXEC CICS GET CONTAINER('DFHFUNCTION')
007200                INTO(WS-DFHFUNCTION)
007300                FLENGTH(length of WS-DFHFUNCTION)
007400                NOHANDLE
007500        END-EXEC.
007600* If not RECEIVE-REQUEST then exit
007700        IF WS-DFHFUNCTION equal 'RECEIVE-REQUEST'
007800            PERFORM VALIDATE-REQUEST THRU END-VAL-REQUEST
007900            PERFORM CHANGE-TRANID    THRU END-CHANGE-TRANID
008000            EXEC CICS
008100                DELETE CONTAINER('DFHRESPONSE')
008200            END-EXEC
008300        END-IF
008400        EXEC CICS RETURN END-EXEC.
008500        GOBACK.
008600*-----*
008700    EJECT
008800*-----*
008900*----- REQUEST VALIDATION -----*
009000*-----*
009100    VALIDATE-REQUEST.
009200        PERFORM GET-SOAP-WEBSERVICE THRU END-GET-SOAP-WEBSERVICE.
009300        IF WS-WEBSERVICES = 'ERROR'
009400            PERFORM FAULT-INVREQ
009500        END-IF.
009600    END-VAL-REQUEST. EXIT.
009700*-----*
009800    EJECT
009900*-----*
010000* Retrieve the DFHWS-WEBSERVICE that contains type of requests
010100*-----*
010200    GET-SOAP-WEBSERVICE.
010300        EXEC CICS
010400            GET CONTAINER('DFHWS-WEBSERVICE')
010500            SET(ADDRESS OF CONTAINER-DATA)
010600            FLENGTH(CONTAINER-LEN)
010700        END-EXEC.
010800* Copy the input container to our storage
010900        MOVE CONTAINER-DATA(1:30) TO WS-WEBSERVICES.

```

```

011000     MOVE CONTAINER-LEN TO WS-WEBSERVICES-LEN.
011100 END-GET-SOAP-WEBSERVICE. EXIT.
011200*-----
011300     EJECT
011400*----- CHANGE DEFAULT TRANID CPIH/CIPQ -----
011500 CHANGE-TRANID.
011600         EXEC CICS GET CONTAINER('DFHWS-TRANID')
011700             SET(ADDRESS OF CONTAINER-DATA)
011800             FLENGTH(CONTAINER-LEN)
011900         END-EXEC.
012800     IF WS-WEBSERVICES = 'placeOrder'
012900         MOVE 'ORDS' TO CA-TRANID
012901         PERFORM CHANGE-CONTAINER THRU END-CHANGE-CONTAINER
012902     END-IF.
013200 END-CHANGE-TRANID. EXIT.
013300*-----
-
013400     EJECT
013500*-----
-
013600 CHANGE-CONTAINER.
013700     MOVE CA-TRANID TO CONTAINER-DATA(1:4)
013800     EXEC CICS PUT CONTAINER('DFHWS-TRANID')
013900         FROM(CONTAINER-DATA)
014000         FLENGTH(CONTAINER-LEN)
014100     END-EXEC.
014200     DISPLAY NN '>-----<'
014300     DISPLAY NN 'Container Name: : DFHWS-WEBSERVICE '.
014400*     DISPLAY NN 'Content length: '    WS-WEBSERVICES-LEN.
014500     DISPLAY NN 'Container content: ' WS-WEBSERVICES.
014600     DISPLAY NN '-----'
014700     DISPLAY NN 'Container Name: : DFHWS-TRANID '.
014800*     DISPLAY NN 'Content length: '    CONTAINER-LEN
014900     DISPLAY NN 'Container content: ' CONTAINER-DATA.
014910         EXEC CICS GET CONTAINER('DFHWS-URI')
014920             SET(ADDRESS OF CONTAINER-DATA)
014930             FLENGTH(CONTAINER-LEN)
014940         END-EXEC.
014950     DISPLAY NN '-----'
014960     DISPLAY NN 'Container Name: : DFHWS-URI  '.
014970*     DISPLAY NN 'Content length: '    CONTAINER-LEN
014980     DISPLAY NN 'Container content: ' CONTAINER-DATA.
015000 END-CHANGE-CONTAINER. EXIT.
015100     EJECT
015200*****

```

```

015300* We detected that the ca_request_id field specifies an invalid
015400* request. This is a CLIENT error.
015500*****
015600 FAULT-INVREQ SECTION.
015700*-----*
015800* Generate a SOAP Fault
015900*-----*
016000     MOVE 'Request code invalid' to WS-FAULT-STRING
016100     EXEC CICS SOAPFAULT CREATE
016200             FAULTCODE(dfhvalue(CLIENT))
016400             FAULTSTRLEN(length of WS-FAULT-STRING)
016500     END-EXEC.
016600 FAULT-INVREQ-END. EXIT.
016700*****
016800* The supplied ca_item_req_ref reference is not in our database
016900* We decide to send a SOAP Fault.
017000* This is a SERVER error.
017100* We supply detailed information in the DETAIL Fault element
017200*****
017300 FAULT-NOTFND SECTION.
017400*-----*
017500* Build the Detail section.
017600* we do it this way for pedagogical reasons
017700*-----*
017800     MOVE ':o( NOT FOUND )o:' to WS-FAULT-STRING
017900     MOVE
018000     '<hdlh:FaultDetail xmlns:hdlh="http://HDRHDLRX.fault">'
018100             to WS-Fault-Namespace.
018200     MOVE '<hdlh:ReturnCode>' to WS-Fault-RC-Lit.
018300     MOVE CA-PARSER-RET-CODE to WS-Fault-RC.
018400     MOVE '</hdlh:ReturnCode>' to WS-Fault-RC-ELit.
018500     MOVE '<hdlh:Item>' to WS-Fault-Item-Lit.
018600     MOVE ca-PARSER-REF-REQ to WS-Fault-Item.
018700     MOVE '</hdlh:Item>' to WS-Fault-Item-ELit.
018800     MOVE '</hdlh:FaultDetail>'
018900             to WS-Fault-ENamespace.
019000     EXEC CICS SOAPFAULT CREATE
019100             DETAIL(WS-Fault-NotFnd)
019200             DETAILLENGTH(length of WS-Fault-NotFnd)
019300             FAULTCODE(dfhvalue(SERVER))
019400             FAULTSTRING(WS-FAULT-STRING)
019500             FAULTSTRLEN(length of WS-FAULT-STRING)
019600     END-EXEC.
019700 FAULT-INVREQ-END. EXIT.

```


Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks publications and Redpaper publications

For information about ordering these publications, see “How to get Redbooks” on page 387. Note that some of the documents referenced here might be available in softcopy only.

- ▶ *Implementing CICS Web Services*, SG24-7657
This book covers the new capabilities of CICS TS V3.2.
- ▶ *Implementing CICS Web Services*, SG24-7206-02
This book contains documented end-to-end Web services security scenarios tested with CICS TS V3.1.
- ▶ *Securing Access to CICS within an SOA*, SG24-5756
- ▶ *Web Services Handbook for WebSphere Application Server 6.1*, SG24-7257
- ▶ *System z Cryptographic Services and z/OS PKI Services*, SG24-7470
- ▶ *WebSphere MQ Security in an Enterprise Environment*, SG24-6814
- ▶ *IBM WebSphere DataPower SOA Appliances Part I: Overview and Getting Started*, REDP4327
- ▶ *IBM WebSphere DataPower SOA Appliances Part II: Authentication and Authorization*, REDP4364
- ▶ *IBM WebSphere DataPower SOA Appliances Part III: XML Security Guide*, REDP4365
- ▶ *IBM WebSphere DataPower SOA Appliances Part IV: Management and Governance*, REDP4366

Other publications

These publications are also relevant as further information sources:

- ▶ *CICS TS V3.2 RACF Security Guide*, SC34-6835
- ▶ *z/OS Security Server RACF Security Administrator's Guide*, SA22-7683
- ▶ *z/OS ICSF Application Programmer's Guide*, SA22-7522
- ▶ *ICSF Administrator's Guide*, SA22-7521
- ▶ *WebSphere MQ Security*, SC34-6588

Online resources

These Web sites are also relevant as further information sources:

- ▶ WS-Trust, see the Web Services Trust Language specification that is published at:
<http://www.ibm.com/developerworks/library/specification/ws-trust/>
- ▶ Specification: Web Services Security (WS-Security) Version 1.0 (April 2002)
<http://www.ibm.com/developerworks/webservices/library/ws-secure/>
- ▶ Web Services Security Addendum (August 2002):
<http://www.ibm.com/developerworks/webservices/library/ws-secureadd.html>
- ▶ Web Services Security: SOAP Message Security V1.0 (March 2004):)
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>
- ▶ The Basic Authentication protocol is specified in RFC2617,
<http://www.ietf.org/rfc/rfc2617.txt>
- ▶ You can obtain an X.509 certificate from a certificate authority such as Verisign, whose Web page is at:
<http://www.verisign.com/>

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Archived

Index

Numerics

2EE application 318
3270 interface 174
3270 terminals 1

A

Advanced authentication 41
Advanced Encryption Standard (AES) 78
AES algorithm 46
alert protocol 121
APAR
 OA14956 48
API(OPENAPI) 146
Application Server Toolkit (AST) 57
Assembler 139
asymmetric cryptography 79
auditing 6
Authenticate 203
authentication 5
authorization 5

B

Base64-encoded ASCII data 213
basic 16
Basic authentication 35
basic authentication 16
BinarySecurityToken 222
Blind trust 58, 260
Blind trust model 59
block cipher 71

C

C 139
CatalogSec_WS-Security.ear 243, 321
CEDA 169
 commands 174
Central Processor Assist for Cryptographic Functions (CPACF) 134
CERTIFICATE 15
Certificate 203
certificate revocation 109
CERTIFICATE_USERID 219

certification paths 107
CEX2C 136
CICS
 as a service provider 7
 as a service requester 13
 CA certificate 190–191
 catalog manager 174
 PI Level 2 Trace 341
 region's keyring 184
 resources in RACF 165
 security 2, 134
 server certificate 187
 Sockets 219
 SSLCACHE 24
 -supplied security handler DFHWSSE1 12
 support for WS-Trust 37
 System initialization parameters related to SSL
 18
 CRLPROFILE=PROFILENAME 19
 KEYRING=keyring-name 19
 MAXSSLTCBS={8 | number} 19
 SIT parameters
 ENCRYPTION 19
 SSLCACHE={CICS | SYSPLEX} 20
 SSLDELAY={600 | number} 20
 Trace 208
 WS-Security Component 36
CICS and SOAP
 message security 35
CICS default user ID 3
CICS region user ID 3
CICS Transaction
 CEDA 169
CICS Transactions 8
 CWXXN 8
CICS user IDs 2
cipher 71
 suite 119
Cipher Block Chaining (CBC) 72
Cipher Feedback (CFB) 72
CIPHERS 15
Ciphers 203
Ciphers in TCPIP SERVICES 203
CIWSMSGH 170

CIWSMSGO 170, 244
 message handler program 245
 CKDS 140
 COBOL 139
 Cobol 17
 Command security 2
 COMMAREA 8
 confidentiality 6
 CONFIGFILE 173
 container 8
 CP Assist for Cryptographic Functions (CPACF)
 135, 228
 CPIH 8
 CRLPROFILE 197
 SIT parameter 198
 Crypto Express 2 Accelerator (CEX2A) 136
 Crypto Express2 (CEX2) feature 132, 228
 Cryptographic Coprocessor Feature (CCF) 134
 Cryptographic Express 2 Accelerator (CEX2A) 134
 Cryptographic Express 2 Coprocessor (CEX2C)
 134
 cryptographic hardware 227
 Cryptographic Service Facility (ICSF) 17
 cryptography 69–70
 hardware 228
 public key 79
 secret key 70
 symmetric 70
 cryptosystem 80
 CSFKEYS 139
 class 140
 CSFSERV
 class 139
 Custom Token 353
 CWXN 8

D

Data 72
 Data Encryption Algorithm (DEA) 72
 Data Encryption Standard (DES) 72
 decrypting 70
 default user ID, CICS 3
 denial of service 52
 DES algorithm 46
 DFH\$ECFN JCL 174
 DFHECAT JCL 174
 DFHPIRT 54
 DFHREQUEST 54

DFHRESPONSE 54
 DFHRPL 48, 243
 DFHWS-AUTHTOKEN 349
 DFHWS-DATA 14
 DFHWS-RESTOKEN 54
 DFHWSSE1 48, 146, 223
 handler 244
 DFHWS-STSE-EP 349
 DFHWS-STSFault 55
 DFHWSTC-V1 54
 DFHWS-TOKEN-NS 349
 DFHWS-TRANID 171, 223, 282, 315
 DFHWS-USERID 12, 41, 282, 315
 DFHWS-WEBSERVICE 8, 170
 Diffie-Hellman 113
 digital signature 70, 93
 Digital Signature Algorithm (DSA) 91
 digital signatures 4
 Distinguished Name 153–154, 157

E

EAR file 272, 321
 eavesdropping 5
 ECFG. 174
 EDSA storage 48
 EDSALIM 48
 EGUI 174
 EJBs 271
 Electronic Codebook (ECB) 72
 Elliptic Curve Digital Signature Algorithm (ECDSA)
 93
 Enabling WS-Trust with TFIM 152
 encrypting 70
 ENCRYPTION 197
 encryption algorithm 119
 error handling 121

F

FTP 216

H

handshake protocol 119, 122
 hardware
 cryptography 228
 hash algorithm 119
 hashing 70, 84
 hashing algorithms 136

HFS
 file 11, 164
HFS file 163, 232
HTTP 3
 Authorization header 16
 basic authentication 17
 endpoint 51
 request 65
HTTP requests 8
HTTP transport 16
HTTPS 14

I
ICSF 227
 callable services 139
 key 303
ICSF callable services 142
 CSNBCKM 142
 CSNBDEC 143
 CSNBENC 143
 CSNBOWH 143
 CSNBRNG 143
 CSNBSYD 143
 CSNBSYE 144
 CSNDDSG 144
 CSNDDSV 144
 CSNDPKB 144
 CSNDPKD 145
 CSNDPKE 145
identification 5
Identity assertion 36
identity assertion 58, 259
Identity assertion with WebSphere for z/OS 155
Integrated Cryptographic Service Facility (ICSF)
229
integrity 6
intermediary 62–63
intermediate gateway
 identity assertion 58, 259
Interoperation with a trusted third party 36
iterated block cipher 71

J
J2EE applications 182
Java 62
Java Authentication and Authorization Service
(JAAS) 336
JCL

DFH\$ECFN 174
JCL DFHECAT 174
JMS 51
JOBLIB 197

K
Kerberos 37
Kerberos Ticket 353
Kerberos Tickets 338
Kerberos tokens 42
key agreement protocol 113
key exchange algorithm 119
key exchange protocol 113
KEYRING 197
 SIT parameter 200
KSDS VSAM 174

L
LINKLIST 197
Linklist command 197
LTPA 42
LTPA Token 353
LTPA token 157
LTPA Tokens 338

M
MACs 136
Mapping Module 338
master secret 125
Maxdatalen 203
MAXSSLTCBS 20, 198
message authentication codes (MAC) 88
MRO 262
mustUnderstand 250, 278, 342

N
National Institute of Standards and Technology
(NIST) 72
non-repudiation 6

O
OPUT 164
Output Feedback (OFB) 72

P
Passticket 37

- performance of message handler 41
- Peripheral Component Interconnect - Extended Cryptographic Coprocessor (PCIXCC) 134
- Peripheral Component Interconnect Cryptographic Accelerator (PCICA) 134
- Peripheral Component Interconnect Cryptographic Coprocessor (PCICC) 134
- PIPELINE 8, 11
 - defining 172
- pipeline
 - configuration file
 - customizing 170
- pipeline alias transaction 10
- PKA 227
 - applications 227
 - master key 227
 - private key 227
- PKA digital signatures 136
- PKCS12 type truststore 190, 235
- PKDS 140
 - key label 227
- PL/I 139
- Portnumber 202
- Protocol 203
- public key cryptography 79
 - cryptography
 - public key 69
- public key infrastructure (PKI) 18

R

- RACDCERT 163
- RACDCERT command 232
- RACDCERT GENCERT 187, 230
- RACDCERT LIST 188
- RACF 153–154, 157, 227
 - authority 185
 - commands 275, 305
 - database 164
 - definitions 151
 - PERMIT 140
 - user registry 260
- Rational® Application Developer (RAD) 57
- RDO 324
 - commands 246
- Redbooks Web site 387
 - Contact us xiv
- request consumer 56
- request generator 56

- Resource security 2
- response consumer 57
- response generator 57
- RSA 51
- RSA algorithm 80
- RSA public key algorithm 136

S

- SAF EJBROLE 271
- SAML 37
- SAML Assertion 353
- SAML assertions 42
- SCEERUN 48
- SDFHWSLD 48, 243, 322
- secret 70
- secret key cryptography 70
 - cryptography
 - secret key 69
- Secure Sockets Layer (SSL) 3
- security
 - auditing 6
 - authentication 5
 - authorization 5
 - confidentiality 6
 - exposures 3
 - HTTP 16
 - identification 5
 - integrity 6
 - non-repudiation 6
 - SOAP message 6
 - see WS-Security
 - transport-level 6, 16
 - comparison with SOAP message security 61
- Security Assertion Markup Language (SAML) 60
- Security exposures 3, 5
- Security Token 3
- Security Token Service (STS) 31, 50
- security tokens 1
- servlet 271
- SETROPTS
 - command 140
- SHA1 51
- signature
 - processing 196, 243, 321
- SIT parameters 161
 - DFLTUSER 3
 - KEYRING 19, 200, 231

- MAXSSLTCBS 19
- SEC 161
- SEC=YES 161
- SECPRFX 162
- SECPRFX=YES 161
- SSLCACHE 20
- SSLDELAY 20
- XTRAN 162, 325
- XTRAN=YES 161
- XUSER 162
- XUSER=YES 161
- SIXMLOD1 48
- SOAP
 - body 8
 - outbound data conversion 8
 - fault messages 253
 - header 155
 - headers 6
 - message 4, 33
 - integrity 42, 45
 - message security 6, 32, 38, 61
 - message security options 38
- Socketclose 203
- spoofing 4
- SSL 194
 - client 13
 - handshakes 17
- SSL/TLS 181
 - CICS support for 17
 - enabling CICS support 18
 - with WMQ 28
- SSL/TLS connection 181
- SSL_DHE_DSS_WITH_AES_128_CBC_SHA 195
- SSL_DHE_RSA_WITH_AES_128_CBC_SHA 195
- SSL_DHE_RSA_WITH_AES_256_CBC_SHA 195
- SSL_RSA_WITH_3DES_EDE_CBC_SHA 195
- SSL_RSA_WITH_AES_128_CBC_SHA 195
- SSL_RSA_WITH_AES_256_CBC_SHA 195
- SSL_RSA_WITH_DES_CBC_SHA 195
- SSL_RSA_WITH_RC4_128_MD5 195
- SSL_RSA_WITH_RC4_128_SHA 195
- SSLCACHE 198
 - SIT parameter 200
- SSLDELAY 20, 24, 198
 - SIT parameter 201
- SSLv2 194
- SSLv3 194
- Station-to-Station (STS) protocol 115
- STEPLIB 197
- STS 315
 - endpoint 51
 - subkey 71
 - surrogate 2
 - surrogate access 247, 275, 305, 325
 - symmetric cryptography 70
 - Symmetric Keys Master Key(SYM-MK) 141
- System Initialization Table (SIT) Parameters 197
 - CRLPROFILE 197
 - ENCRYPTION 197
 - KEYRING 197
 - MAXSSLTCBS 198
 - SSLCACHE 198
 - SSLDELAY 198
- System SSL libraries 197

T

- TAM Credential 353
- tampering 4
- tbsCertificate 99
- TCP/IP Monitor 207, 248
- TCPIPSERVICE 8–9, 18, 196
 - attributes
 - AUTHENTICATE 16
 - PORTNUMBER 169
 - PROTOCOL 169

TFIM

- administrator 336
- Configuring the WebSphere service requester 318
- Integrated Solutions Console 328
- Preparation for this scenario 316
- Request from CICS to TFIM 343
- Scenario overview 314
- Testing the WS-Trust scenario 341
 - to map user identity data 157
- TFIM WS-Trust enable 152
- TFIM-WS-Trust 313
- Tivoli Federated Identity Manager 37
- Tivoli Federated Identity Manager (TFIM) 313
- Tivoli Federated Identity Manager, 31
- TLS 194
- TLSv1 194
- Transaction security 2
- transport 61
 - security 16
- Transport Layer Security (TLS) 116
- Transport-level security 6

Trust Chain 326, 328
trust token 58, 260
trusted 79
Trusted authentication 41

U

URIMAP 8–9, 14–15, 64, 185
UserName Token 353
UsernameToken 40, 53
UsernameTokenGenerator 268
UsernameTokens 37

V

verification 92
VSAM
 datasets 174

W

Web service 12
Web services atomic transactions (WS-AT) 63
Web Services Distributed Management (WSDM)
60
Web Services Trust Language 37
Web Services Trust Language (or WS-Trust) 31
Web Services-Security road map 34
WEBSERVICE 9, 12
 attributes
 WSBIND 12
 WSDLFILE 12
WebSphere 55
 CA Certificate 163
 keystore 228
 truststore 228
WebSphere Application Server Administrative
 Console 181
WebSphere CA certificate 163
WebSphere CA certificates 228
WebSphere DataPower 281
 Configuration 286
 Configure Crypto Certificate 291
 Configure the Crypto Validation Credentials
 293
 Scenario overview 282
 Testing the Identity assertion 306
 Using the Probe facility of DataPower 308
WebSphere MQ 16
WS Binding page 268

WS Extension page 266
WS Security wizard 265
WS-Authorization 35
wsbind files 151
WSDIR 173
WS-Federation 35
WS-Policy 35
WS-Privacy 35
wsse_handler 53
WS-Secure Conversation 35
WS-Security 32, 34
 configuration
 files 207, 246–247, 274–275, 305, 324
 message authentication 38
 specifications 34
 test scenario 207, 247, 275
 typical scenario 33
WS-Security Component 36
WS-Trust 35
WS-Trust handler 343
WS-Trust with TFIM 313

X

X.509 certificate 17, 36, 222
X.509 certificate authentication 41
X.509 certificates 3
X.509 digital certificate 49
X.509 distinguished name 187
X.509 token 59
X.509 token profile 40
X.509 tokens 37
X509 certificate 156, 281
X509 Token 353
XML
 digital signature processing 226
 digital signatures 44
 elements 32
XML digital signature 33, 57, 153, 156, 238, 319
XML encryption 32, 57, 238, 319
XML Key Management Specification (XKMS) 60
XML Schema Definition (XSD) 60
XML Stylesheet Language Transformation (XSLT)
60
XSLT
 file 336

Z

z/OS 228

z/OS address spaces 262
z/OS system 262
z10 (Enterprise Class (EC) 135
z9 Business Class (BC) 135
z9 Enterprise Class (EC) 135
z900 136

Archived

Archived



Securing CIGS Web Services

(0.5" spine)
0.475" <-> 0.875"
250 <-> 459 pages

Archived



Securing CICS Web Services

WS-Security and SSL/TLS support in CICS TS V3.2

Interoperability with WebSphere DataPower

Using Tivoli Federated Identity Manager

Securing access to information is important to any business, especially for business-critical systems that manage sensitive data, as is often the case for systems based on IBM Customer Information Control System (CICS). Security becomes even more critical for implementations structured according to service-oriented architecture (SOA) principles, due to loose coupling of services and applications, and their possible operations across trust boundaries.

In this IBM Redbooks publication, we consider the different ways that CICS Web services can be secured. We consider transport-level security mechanisms such as SSL/TLS and CICS support for the message-based security specifications WS-Security and WS-Trust.

To assist solution and security architects, we outline the main planning considerations and make recommendations on the choice of a security solution. For the systems programmer, we provide detailed setup guidance for configuring common security scenarios. These scenarios include interoperability with WebSphere DataPower and using Tivoli Federated Identity Manager (TFIM) as a Security Token Service.

For each scenario, we provide step-by-step configuration information for CICS and the other involved systems, including WebSphere Application Server, WebSphere DataPower, and TFIM.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks