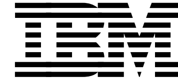
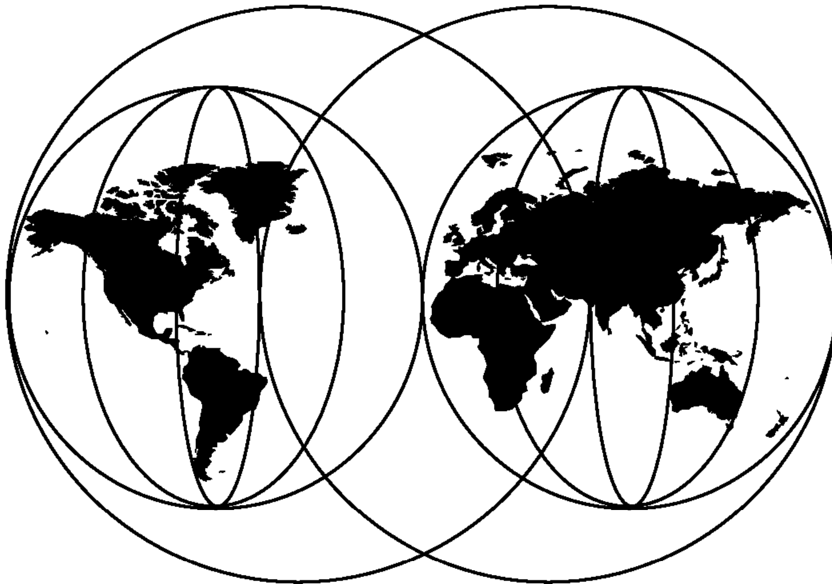


MQSeries Security: Example of Using a Channel Security Exit, Encryption and Decryption

Dieter Wackerow, David Shogren
Manfred Lotz



International Technical Support Organization

http://www.redbooks.ibm.com

This book was printed at 240 dpi (dots per inch). The final production redbook with the RED cover will be printed at 1200 dpi and will provide superior graphics resolution. Please see "How to Get ITSO Redbooks" at the back of this book for ordering instructions.

SG24-5306-00

SG24-5306-00

International Technical Support Organization

MQSeries Security: Example of Using a Channel Security Exit, Encryption and Decryption

December 1998

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix B, “Special Notices” on page 89.

First Edition (December 1998)

This edition applies to MQSeries for Windows NT Version 5.0.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HZ8 Building 678
P.O. Box 12195
Research Triangle Park, NC 27709-2195

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1998. All rights reserved.**
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Figures v

Preface vii
The Team That Wrote This Redbook vii
Comments Welcome viii

Chapter 1. Secure Communications - A Beginner′s Guide 1
1.1 Terminology 1
1.2 Goals of Secure Communication 3
1.3 Cryptographic Algorithms 3
1.3.1 Secret Key Cryptography 4
1.3.2 Public Key Cryptography 5
1.3.3 One-way Hash Functions 6
1.3.4 Random Sequence Generator 6
1.4 Basics of Cryptographic Protocols 7
1.4.1 Digital Signature 7
1.4.2 Public Key Certificate 8
1.4.3 Key Distribution and Certificate Authority 8

Chapter 2. About Channel Exits 11

Chapter 3. Design of a Security Exit 17
3.1 Overview 17
3.2 Cryptographic Software 18
3.3 Channel Security Server Database (CSSD) 18
3.4 Abbreviations Used in This Chapter 19
3.5 Handshaking 19
3.6 Sending a Message 20
3.7 Client Applications Using the Channel Security Exit 21
3.8 The Channel Security Exit as Part of a Security Concept 21

Chapter 4. Implementation of a Security Exit 23
4.1 Overview 23
4.2 CSSD 23
4.3 Handshaking 24
4.4 Messages 25

Chapter 5. Channel Security Exit on Different Platforms 27
5.1 How to Compile and Link a Channel Security Exit on Windows NT . 27
5.2 How to Install the Channel Security Exit on Windows NT 28
5.2.1 CSSD 28

© Copyright IBM Corp. 1998

5.2.2 How to Activate the Channel Security Exit 28
5.3 How to Set Up Channel Definitions 29
5.3.1 How to Define a Sender/Receiver Channel Pair 29
5.3.2 How to Define a SVRCONN/CLNTCONN Channel Pair 30
5.4 How to Port the Exit to AIX 30
5.5 How to Port the Exit to OS/390 31

Appendix A. MQSeries Channel Security Exit Source Code 33
A.1 Header File MQCHEXIT.H 33
A.2 Header File MQSEC.H 34
A.3 Source Code of CSSD.C 37
A.4 Source Code of MQCHEXIT.C 47

Appendix B. Special Notices 89

Appendix C. Related Publications 91
C.1 International Technical Support Organization Publications 91
C.2 Redbooks on CD-ROMs 91
C.3 Other Publications 91

How to Get ITSO Redbooks 93
IBM Redbook Fax Order Form 94

List of Abbreviations 95

Index 97

ITSO Redbook Evaluation 99

Figures

1. Communication without Encryption 1
2. Communication with Encryption 2
3. Communication with Encryption Using a Key 2
4. Channel Concepts 11
5. Channel Types 12
6. Channel Exits 13
7. Handshaking Communication Flow 24
8. MQCHEXIT.H 33
9. MQSEC.H 34
10. CSSD.C 37
11. MQCHEXIT.C 47

© Copyright IBM Corp. 1998

Preface

This redbook describes how to achieve secure communications in an MQSeries environment using security exits, send message and receive message exits.

This redbook provides an example of an exit running in Windows NT. The exit can be used in MQSeries node to MQSeries node communications as well as for client applications connected to an MQSeries server.

The sample exit uses a real-life example of cryptography software for performing encryption and decryption.

The first two chapters provide an overview of what secure communications is all about and an introduction to channels and channel exits in general. The next two chapters describe the design of a channel security exit and its actual implementation. Chapter 5 is more technical and describes:

- How to compile and link the channel security exit
- How to install the exit on your system
- How to set up channel definitions for using the exit
- How to port the application to AIX and OS/390

The appendix contains the source code of the channel security exit as well as the source code of the program to generate the private and public key files.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.

The authors of this redbook were:

Manfred Lotz
Deutsche Boerse Systems, Frankfurt, Germany
Manfred_Lotz@exchange.de

David Shogren
Dieter Wackerow
International Technical Support Organization, Raleigh Center

© Copyright IBM Corp. 1998

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

John Barbesgaard, IBM Denmark
Cristiane Petrillo Hilkner, IBM Brazil
Michael Leijdecker, IBM Netherlands
Bettina Rockmann, APCON Professional Concepts GmbH, Hannover, Germany
Carla Sadtler, International Technical Support Organization, Raleigh Center

Comments Welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in "ITSO Redbook Evaluation" on page 99 to the fax number shown on the form.
- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Send your comments in an Internet note to `redbook@us.ibm.com`

Chapter 1. Secure Communications - A Beginner′s Guide

This chapter discusses the basics of secure communications. It is not meant to replace an introductory text about the subject. The intention is to provide an introduction to enable the reader to understand how communications can be made as secure as possible and to understand how this can be put into effect in an MQSeries environment using an MQSeries channel security exit.

1.1 Terminology

If a sender sends a message to a receiver without doing any preventive actions an eavesdropper could read the message. The eavesdropper could modify the message in some way to the disadvantage of either the sender or the receiver or both of them and nobody would even notice.

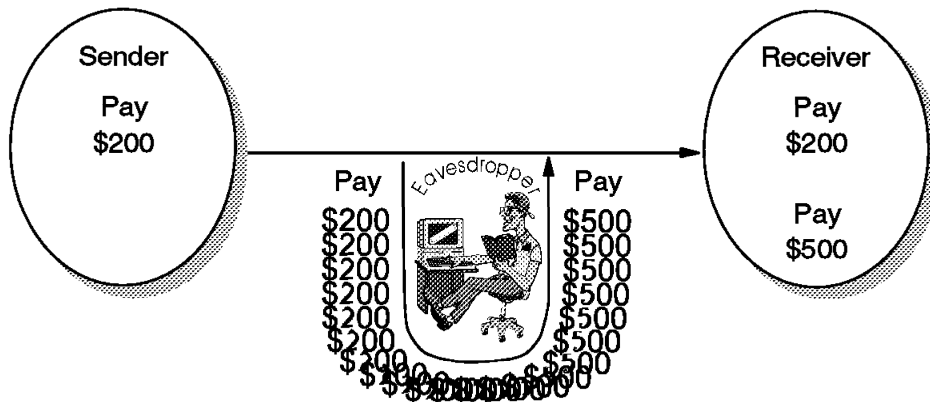


Figure 1. Communication without Encryption

To prevent this from happening a message has to be changed in a specific way to hide its original content. This method is called *encryption*. The encrypted message is called *ciphertext*. The original message is called *plain text*. The method to turn an encrypted message back into its original state, the plain text, is called *decryption*.

In general the method of encrypting a plain text and decrypting a ciphertext is called *cryptographic algorithm* which is a predefined set of steps to carry out encryption and decryption.

© Copyright IBM Corp. 1998

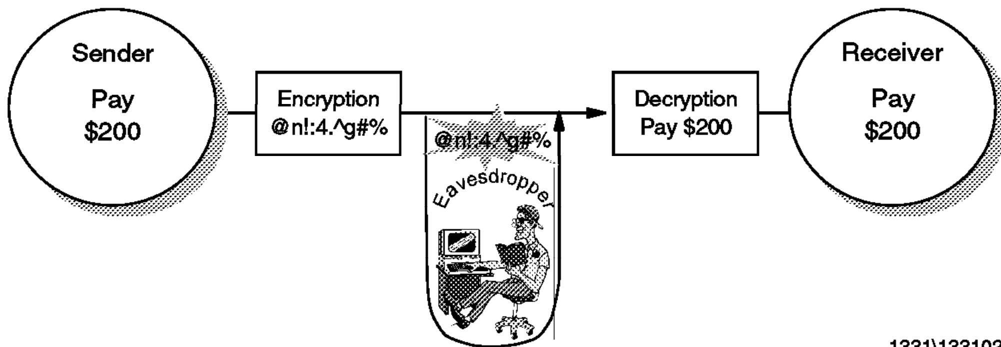


Figure 2. Communication with Encryption

To make sure a cryptographic algorithm doesn't need to be changed because for example a member of a department leaves the company, a key is used to encrypt and decrypt data. A key is a number of values which will be used to encrypt plain text and to decrypt ciphertext. For example, a password used for logging in is a key.

We use the following notation to denote encryption and decryption:

$E_K(M) = C$ Means a plain text message M is encrypted with the key K to get the ciphertext C.

$D_K(C) = M$ A ciphertext C is decrypted with a key K to get the plain text message M.

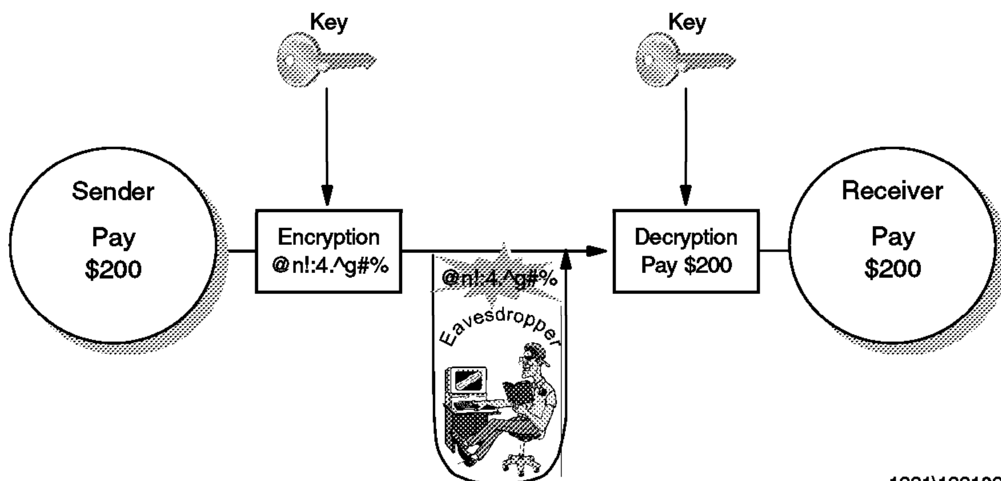


Figure 3. Communication with Encryption Using a Key

1.2 Goals of Secure Communication

Secure communication should fulfill the following objectives:

Non-repudiation The author of a message cannot deny authorship of the message he sent.

Integrity It can be verified that the message has not been modified in any way during its way to the receiver.

Authentication The message comes from where it claims to come.

Confidentiality A message can only be read by the receiver.

We will see later in this chapter that a message encrypted with a secret key together with a digital signature can best fulfill the objectives mentioned here.

The method to use when establishing secure communications is *cryptography*.

Security consists of much more than cryptography. The best cryptographic algorithms won't help if there are weaknesses in other areas, because security is only as good as the weakest link in the chain.

Potential threats among others are:

- Users often choose poor passwords, which makes even the best cryptographic algorithm vulnerable against the so-called *dictionary* attack which means the attacker simply tries to break a code by using words from a dictionary. Sometimes a user even writes down a password and keeps the note in a place easily accessible to others.
- Existing security software is not used consistently. For example, an important file is not protected and can act as a backdoor for a potential attacker.
- Organizational problems. For example, unauthorized people are able to access equipment that should be locked away.

1.3 Cryptographic Algorithms

Cryptographic algorithms use keys to encrypt plain text. The idea is to make decryption easy for the party who owns the key and hard for anybody else. Hard means that to break a key takes so long that the information is no longer of any use to the attacker.

In cryptography many different mathematical areas are involved such as number theory, complexity theory, information theory, algebra of finite fields and statistics. Other factors, also, are involved.

1.3.1 Secret Key Cryptography

A secret key algorithm uses a secret key for both encryption and decryption. Sender and receiver have to agree upon a secret key prior to the start of any communication.

If the key is compromised, then anybody can get access to the data encrypted with that particular key.

We use the following notation to denote encryption and decryption using secret key cryptography.

$E_S(M) = C$ Means a plain text message M is encrypted with the secret key S to get the ciphertext C.

$D_S(C) = M$ A ciphertext C is decrypted with the secret key S.

There are two categories of algorithms to implement secret key cryptography. The first category consists of algorithms operating on the plain text a single byte at a time. They are called *stream ciphers*. The second category are algorithms working on a group of bytes at a time, called *block ciphers*.

Only block ciphers are of interest because they give far better results in terms of encryption quality.

A typical blocksize is 64 bits, which provides efficient software or hardware implementations and resists attacks very well.

For over 20 years the worldwide standard has been DES (Digital Encryption Standard), which was developed by IBM in the 1970s. The security of DES has been questioned quite often in the past 20 years. One allegation is that the NSA (National Security Agency) was involved in some obscure way during the development of the algorithm, which made people suspect that they implemented a trapdoor in the algorithm. The other reason was that due to the constantly increasing processing power of today′s computer, a key of 56 bits no longer seems to be safe enough.

For example, in 1993 scientists showed that it is possible to build a $1 million machine that is able to find a key in an average of 3.5 hours.

In the beginning of the 1990s a new algorithm was invented: IDEA (International Data Encryption Standard). This algorithm was based on the

newest research knowledge available and uses a 128-bit key. Today IDEA is widely regarded as the most secure secret algorithm available to the public.

1.3.2 Public Key Cryptography

In public cryptography the key used for encryption is different from the key used for decryption. The encryption key is made public and the decryption key is the private key accessible only to its owner.

The decryption key cannot be calculated in a reasonable period of time. Any person knowing the public key can encrypt a message but only the person who owns the private key can decrypt it.

We use the following notation to denote encryption and decryption using public key cryptography:

To keep notation simple, we use the same letters for the private and public keys. Using K in the context of function E always means we are referring to the public key and using it in the context of function D means we are referring to the private key.

$E_K(M) = C$ Means a plain text message M is encrypted with the public key K to get the ciphertext C.

$E_K(C) = M$ A ciphertext C is decrypted using the public key K to get the plain text message M. This can only work if encryption was done using the private key.

$D_K(C) = M$ A ciphertext C is decrypted with the private key K.

$D_K(M) = C$ A plain text M is encrypted using the private key.

Notes:

1. Function D can also be used for encryption if applied to plain text.
2. Function E can also be used for decryption if applied to ciphertext.

Today there are three public key algorithms known to be working well on both encryption and digital signatures: RSA, ElGamal and Rabin.

RSA (named after the three inventors Ron Rivest, Adi Shamir and Leonard Adleman) is by far the most well known public key algorithm. RSA gets its security from the fact that it is very hard to factor large numbers.

Public key algorithms are typically very slow compared to secret key algorithms. One reason for this is that public key algorithms have to use far larger keys to achieve the same grade of security as secret key algorithms. For that reason, they are not used to encrypt a large amount of data.

Therefore, the approach most often used is to encrypt a chunk of plain text using a secret key and sending the secret key encrypting it using the public key of the receiver. Then the receiver can retrieve the secret key by decrypting the ciphered secret key. Now he can decrypt the message using the secret key.

1.3.3 One-way Hash Functions

One-way hash functions (or *hash* for short) are checksum-like functions over a portion of data. They are also called *message digest* or *fingerprints*.

Hashes are used to verify that data transmitted to a receiver remains unaltered.

The following criteria apply for hashes:

- A hash is a one-way function in the sense that it is impossible to get back to the data that was used to create the hash.
- Even a single bit change in the input data produces a totally different hash value of that data.
- A good hash makes it nearly impossible to produce two different streams of data with the same hash value.

MD5 developed by Ron Rivest, is a widely used hash function that produces a 128-bit hash.

1.3.4 Random Sequence Generator

Sometimes it is necessary to produce a sequence of random numbers. The problem with computers is that they cannot produce a sequence of truly random numbers. This is because a computer is deterministic and so is the produced sequence of random numbers.

The best that computers can do is to produce a sequence of so-called pseudo random numbers. This is a sequence of numbers where the period is long enough so that the application believes that the sequence is random.

If for example a computer creates a sequence like this: 3, 77, 41, 56, 93, 16, 3, 77, 41, 56, 93, 16, 3 etc. then we see at once that the sequence has a period of six, meaning the seventh number is the same as the first number in the sequence.

If an application would need two numbers, then it wouldn't notice that the sequence is not random. If, on the other hand, the application needs 10 numbers, then the sequence produced in this example would be of no use for that application.

In cryptography a random sequence generator has to be far more sophisticated.

There are statistical procedures to check the quality of a random sequence generator.

1.4 Basics of Cryptographic Protocols

A protocol is a set of steps to define communications between at least two parties. The steps have to be done in a well defined order and there has to be a defined action for all cases coming up during that communication.

1.4.1 Digital Signature

A digital signature is a means of signing a message electronically. Unlike a handwritten signature a digital signature:

Is unforgeable
: The signer and no one else signed the document.

Is not reusable
: The digital signature belongs to a message in a way that it cannot be used to sign a different message.

Cannot be repudiated
: The signer cannot claim later that he didn't sign the document.

To produce a digital signature we use the techniques of secret cryptography and public cryptography.

Here's how it is done:

Assume a message M is about to be sent. A hash of that message, H(M), is calculated. The message itself is encrypted with a secret key. Now H(M) and S are encrypted by using the sender's private key. Finally, the result is encrypted using the receiver's public key.

The message M is sent encrypted like this:

S(M)

And the digital signature is appended like this:

$$DS = E_B(D_A(S+H(M)))$$

Here D_A is the sender's private key, E_B the receiver's public key and S is the secret key. This sometimes is called a digital signature with encryption. In

the following, when saying digital signature, we always mean digital signature with encryption if not otherwise stated.

To decrypt a message the receiver proceeds as follows. First he decrypts the digital signature DS with his own private key. Then he decrypts the result with the sender's public key. Now he decrypts the message M with the secret key S found in the digital signature.

Finally he calculates the hash over the plain text message M and compares this with the hash inside the digital signature. If all goes well he is sure the message came indeed from sender A and the message has not been altered on its way.

1.4.2 Public Key Certificate

For key distribution, keys are stored in a special way called *public key certificate*. A public key certificate consists of the following three parts:

- A user name
- The user's public key
- A digital signature computed over those two using the public key of the receiver of the public certificate

A certificate is signed by a trusted third party called a certificate authority (CA). Signed means the certificate is encrypted with the private key of the certificate authority so that anybody decrypting the public certificate knows for sure it came from the CA.

1.4.3 Key Distribution and Certificate Authority

Key management is one of the most vulnerable parts of cryptography. Basically there are two problems with key management: how to manage the keys and how to deploy them.

A key chosen by a person sitting in front of a computer terminal often is easy to guess because people tend to be lazy in choosing passwords. This makes, for example, a dictionary attack very likely to succeed. In public key cryptography there is a need to deploy public keys. This can be done in a variety of ways. One possibility is to transmit it electronically. However, this gives an eavesdropper a chance to replace it with his own public key. Therefore a way to make this feasible is to sign a public key. This has to be done by a trusted third party. Assume, for example, Adam and Eve want to exchange their public keys and both trust Abel. We assume that Adam as well as Eve know the public key of Abel and vice versa. Then Abel could sign the public key of Adam with his own private key and send it to Eve and

sign the public key of Eve and send it to Adam. Then both Adam and Eve know they received it from Abel and nobody else.

Another way to distribute a public key is to submit it to the requestor manually, for example via diskette.

There could be a centralized database, a so-called *public certificate* where all keys are stored and where all clients get keys. Distributed keys are then signed with the private key of the CA.

Chapter 2. About Channel Exits

This chapter provides some general information about MQSeries distributed messaging and explains what channel exits are and their purpose.

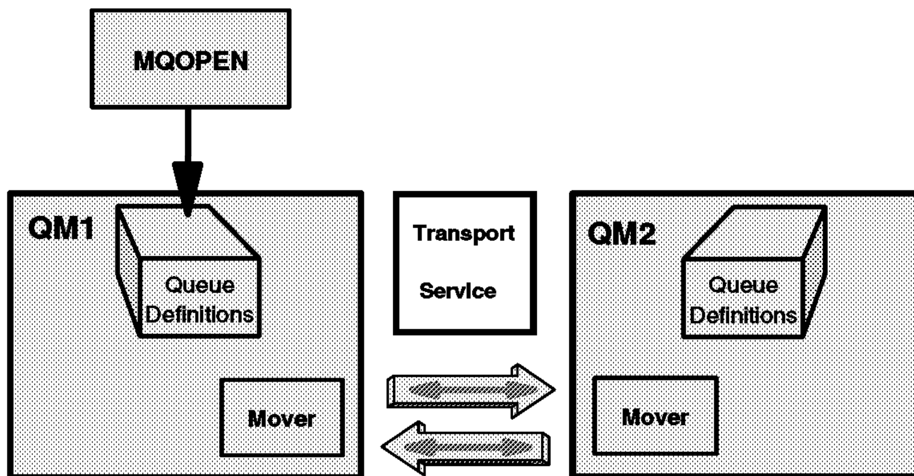


Figure 4. Channel Concepts

In MQSeries, applications communicate via queues. An application opens a queue (just like you would open a file) and puts messages in it. The mover or MCA (message channel agent) is the MQSeries program that moves the messages to the target queue manager using existing transport services. The target queue manager may reside in the same or in another machine. This is transparent to the application; the MQSeries administrator provides definitions that contain routing information.

Figure 4 shows two uni-directional channels connecting two queue managers, QM1 and QM2. Within a channel, data flows only in one direction, while control packets flow in both directions (handshaking).

There are two kinds of channels, message channels and MQI channels. MQI channels connect clients with the queue manager in their server while message channels connect two queue managers. The channel types are:

- Sender
- Server
- Receiver
- Requester
- Client connection
- Server connection

© Copyright IBM Corp. 1998

The latter two are MQI channels. MQI channels are bi-directional. A channel is defined using one of the above types defined at one end, and a compatible type at the other end. Possible combinations are:\

- Sender - receiver
- Requester - server
- Requester - sender (callback)
- Sever - receiver
- Client connection - server connection

For detailed information about channels refer to *MQSeries Intercommunication*, SC33-1872 and *MQSeries Clients*, GC33-1632.

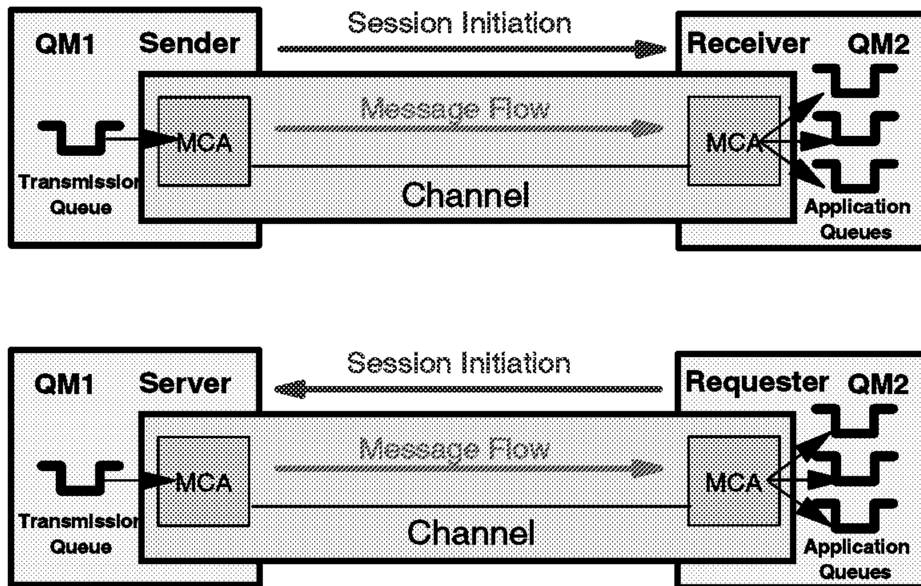


Figure 5. Channel Types

Figure 5 shows two channel pairs, a sender/receiver pair and a requester/server pair. The difference is how the session between the two queue managers is initiated.

QM1 shows a transmission queue. All messages destined for one queue manager are put in this queue, regardless of what program will process them. By convention, the transmission queue name is the same as the target queue manager name.

The MCAs or movers are MQSeries applications that transmit messages from the transmission queue of the sending queue manager to one or more application (input) queues belonging to the target queue manager. A pair of MCAs is known as a channel.

When the target system has not been started or when the target queue manager is not running the messages stay in the transmission queue of the sending queue manager until transmission can be established, either automatically of manually.

We said before that a channel consists of two compatible message channel agents. MQSeries channels provide six exit points that enable a channel to be customized. The channel exits are shown in Figure 6.

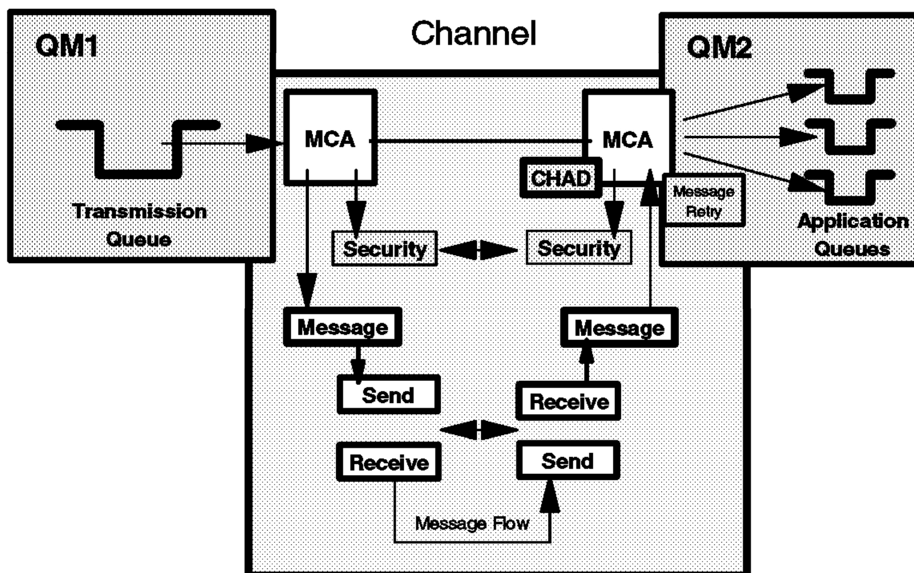


Figure 6. Channel Exits

Note: With MQSeries Version 5 you can chain message exits, send exits and receive exits.

A brief overview of the channel exit types follows:

Security exit
: This exit is used for authentication of the partner MCA when two MCAs connect to one another. The exit is invoked at MCA initiation and termination, and at channel startup after initial data negotiation, but before any user data is exchanged.

The channel exit routines operate as pairs and any exit must be able to terminate the connection. The server or receiver initiates the security message exchange. The sender or requester processes the received security message or initiates a security exchange when the remote end cannot. For more information refer to 4.3, "Handshaking" on page 24.

Note: The name security exit is misleading. It is not only for security. Also, not all security functions need to be performed here. Data exchange is arbitrary. Therefore, any function not related to security may be implemented.

A security exit does not have access to message data and, therefore, cannot be used for functions such as encryption.

Message exit
: A message exit is invoked once per message on either side of the channel, that is, after the message is taken off the transmission queue and before the message is put on the target queue. It can be used for:

- Encryption on the link
- Validation of incoming user IDs
- Message data conversion
- Journaling
- Reference message handling
- Substitution of user IDs according to local policy

In the message exit, you have access to the transmission header, the message header and the data.

Send and receive exits
: These exits are called once per message segment. The data available to the exit program is part of the message and an 8 byte header reserved for the MCA. The exit must have a good response time or the MCA abends. It can be used for data compression and decompression or data encryption and decryption.

The example in this book uses send and receive exits for data encryption and decryption.

Message retry exit
: You can write an exit program that is invoked when an attempt to open a target queue is unsuccessful. You can use it to determine how often to retry or redirect the message to an

alternate queue. The program is invoked at the receiving end of the channel when the MCA is initiated and terminates.

Channel auto-definition exit
With MQSeries Version 5 you don't have to define channels of the types receiver or client connection. MQSeries defines them for you if CHAD (channel auto-definition) is enabled. This exit can be used to modify the default channel definitions.

The following table shows what exits are valid for what channel type and what data is available to the exit routine.

Table 1. Channel Exits

Exit	Channel type						Chaining	Buffer data MQXCP +
	Sender	**Server**	**Receiver**	**Requester**	**CLNTCONN**	**SVRCONN**		
Security	√	√	√	√	√	√		Security message
Message	√	√	√	√			Y	MQXQH + message
Send	√	√	√	√	√	√	Y	8 bytes + data
Receive	√	√	√	√	√	√	Y	8 bytes + data
Retry			√	√				MQXQH + message
CHAD			√			√		Default defintion

Notes:

1. MQXCP = Channel exit parameter structure
2. MQXQH = Transmission queue header and MQMD
3. MQMD = Message descriptor (header)
4. Security message is defined by user, refer to 4.3, “Handshaking” on page 24

Chapter 3. Design of a Security Exit

This chapter describes the design of a channel security exit and also gives some remarks of what to consider, from a security point of view, when designing MQSeries applications.

3.1 Overview

Basically our MQSeries channel security exit provides MQ node to MQ node security. It is totally independent of any application design. It is activated or deactivated by changing a channel definition accordingly.

We have a special case when client applications are involved. In this case a client application is regarded as an MQSeries server.

Each channel security exit uses a pair of public and private keys to verify itself during an initial handshaking phase with its partner. The same applies for the channel security exit on the client application′s side.

All security related information is stored in a central security server database (CSSD). It contains all queue manager names in case of MQ node to MQ node communications or application names in case of client applications connected to a queue manager. It also contains all public and private keys and all channel names involved in MQSeries communications using the channel security exit.

A channel exit needs the public key of its partner to be able to verify its partner. It retrieves the public key of its partner either from an accompanying public key file or if this file doesn′t exist it retrieves the information from the CSSD by using MQSeries communication.

After successfully completing the handshaking procedure message transfer begins.

Encryption and decryption

Encryption and decryption during the message transfer is handled by the channel send and receive exits. Channel message exits cannot be used because they cannot be invoked from a client application.

The following sections describe, in detail, how the various phases are handled and also discusses some important issues to consider when writing an exit.

© Copyright IBM Corp. 1998

3.2 Cryptographic Software

If using a channel security exit it is very important to use cryptographic software which uses the best algorithms available to the public.

Cryptographic software has to be updated constantly to reflect the newest research knowledge.

So don′t be tempted to write your own cryptographic software because the result is either worse than buying a package or it is more expensive to do this. The cryptography software used in this project is RSAEURO V1.04 by by Reaper Technologies. It is available for download from the Internet at http://www.reapertech.com.

3.3 Channel Security Server Database (CSSD)

In this section, we describe why we use a CSSD, what information is stored in a CSSD and in what way an application can interact with the CSSD.

To enable a channel exit to authenticate itself during the initial handshaking it has to know its private key and the public key of its partner to be able to verify its partner.

As described in 1.4.3, "Key Distribution and Certificate Authority" on page 8, one way to distribute keys is via a central database which acts as a Certificate Authority (CA). This is done by the CSSD.

The queue manager′s private and public keys are stored in a private key file which also contains the CSSD′s public key.

To be able to communicate with a partner a security exit has to know the partner′s public key. The channel security exit has two ways to receive this information:

1. It has a public key file associated with its queue manager. This file contains all public keys of all queue managers who also have a security exit running and who can be reached directly.
2. Or, if this file doesn′t exist it sends an MQSeries message to the CSSD requesting the public key information which then comes back via a MQSeries message. These messages are sent via special queues, and for security reasons they are sent encrypted including a digital signature to prevent any forgery.

This database is the most sensitive part of the MQSeries secure environment. To compromise the CSSD means to compromise the entire environment.

3.4 Abbreviations Used in This Chapter

In the following we use some abbreviations which we define below:

A Name of the sending queue manager

B Name of the receiving queue manager

M Message to be sent

H Hash

E_A,E_B Public key of queue manager A or B

D_A,D_B Private key of queue manager A or B

S Generated random session key which is a secret symmetric key to be used to encrypt the message to be sent.

C_M Public certificate of Message M. C_M consists of three parts:

- **A** The sending queue manager′s name
- **S** The secret key generated for this session
- **H(S(M))** The hash value of the encrypted message M (with S encrypted).

3.5 Handshaking

Handshaking gives queue manager A confidence that it is talking to queue manager B and vice versa.

Handshaking is initiated by the receiver who sends a *HAM* (Handshaking Authentication Message) to the sender′s channel security exit to verify itself.

The sender′s channel security exit itself builds a *HAM* as a reply which is returned to the receiver′s side where it is verified.

If on one side there is no channel security exit or the partner exit is not able to authenticate itself then the channel will be closed.

Encryption for all further communication will be done by a secret key which will be generated by the receiving queue manager′s channel security exit. Henceforth we call S the secret session key.

The *HAM* consists of the following parts:

S(M) The message M encrypted with the secret session key. The message itself is an eye catcher concatenated with a 128-bit random key generated by the receiving queue manager.

$D_B(C_M)$ The digital signature encrypted with the receiving queue manager′s private key.

These parts are encrypted using the public key of the receiving queue manager.

The sending queue manager receiving the HAM decrypts the message using its private key.

It retrieves the public key of the receiving queue manager E_B from the MQ Security Database. With the public key it decrypts the digital signature to get the public certificate, $E_B(D_B(C_M)) = C_M$. With the secret session key gotten from the public certificate it decrypts the message S(M) to get M. It looks if the message begins with the eye catcher to make sure the decryption resulted in something meaningful.

If all checks are successful it builds its own *HAM* to authenticate itself to the receiving queue manager. The *HAM* consists of the following parts:

S(M) The message M encrypted with the secret session key. The message itself is the same that was sent by the receiving queue manager.

$D_A(C_M)$ The public certificate encrypted with the sending queue manager′s private key.

3.6 Sending a Message

The data sent to the other queue manager consists of:

S(M) The encrypted message using the secret session key S.

$D_A(C_M)$ The public certificate encrypted with the sending queue manager′s private key.

On the receiving side the security channel exit retrieves the public key E_A from its .PUB file. Then it decrypts the public certificate with $E_A(D_A(C_M)) = C_M$ to get the public certificate in plain text. It decrypts the message S(M) using the secret session key by applying S(S(M)).

3.7 Client Applications Using the Channel Security Exit

A client is treated like a queue manager in the sense that a client application is associated with a pair of private and public keys.

However, it is also possible to assign unique pairs of private and public keys to each user of a client application.

This concept might be extended by letting the client verify itself using user ID and password.

If a client application has to communicate with a legacy application using MQSeries, consider that there may be an existing security concept used in the legacy application. It might be difficult to integrate the existing security concept into an MQSeries channel security exit design. A possible way of integration could be to use the existing user ID and password. A second possibility is to have another user ID and password pair to verify itself to the channel security exit. Either possibility might create a nightmare thereafter because we will have two security systems to maintain.

3.8 The Channel Security Exit as Part of a Security Concept

It is very important to view an MQSeries channel security exit as part of a security concept. It has to fit in a system containing firewalls, application gateways, organizational policies and procedures.

In the following we give some examples of how MQSeries communication can be set up using a channel security exit.

If we have an MQSeries server connected to another MQSeries server belonging to the outside world it is recommended to use a dedicated machine as an MQSeries application gateway. The partner on the other side should do the same. Then both partners have to agree about the channel security exit to use.

If customers use an MQSeries client application it has to be considered if each user has its own unique pair of public and secret key files or if a user has to show a user ID and password each time he logs in. Instead of letting client applications talk directly to MQSeries it might be a better idea to design an application to run as a client under a Web browser and talk to a Web server. The Web server is connected to the MQSeries Server.

Chapter 4. Implementation of a Security Exit

This chapter describes the implementation of a security exit. This is not a full implementation of the design discussed in Chapter 3, "Design of a Security Exit" on page 17.

4.1 Overview

To make this a real world example we used a real cryptographic software package to do encryption, decryption, etc.

The cryptography software used for our channel security exit is called RSAEURO V1.04 by Reaper Technologies and, as of the writing of this book, is available for download from the Internet at http://www.reapertech.com. Refer to the licence agreement included in the package as well as to the regulations concerning the use of cryptographic software in your country.

The security exit is responsible for doing the initial handshaking with its partner security exit. When sending messages the channel send exit takes over to encrypt the data. On the other end the channel receive exit takes over to decrypt the data. The channel message exit could not be used for that task because it is not available for client applications.

We specify the same exit routine name for channel security, channel send and channel receive exit. This way the channel definitions are easier because no confusion can come up over what name to use where. The main routine channel exit acts as a dispatcher to call the appropiate exit routine.

4.2 CSSD

Our current implementation of the CSSD uses two input files to generate the appropriate key files for the participating queue managers.

The first input file contains the name of all queue managers that participate in secure communications. The second file contains all channel names used by our channel security exit with accompanying queue manager names.

Currently we haven't implemented the CSSD as a full featured database. MQSeries communications to retrieve public keys from the CSSD isn't implemented either.

4.3 Handshaking

The channel security exits on both sides are responsible for the initial handshaking. Either side can check if its partner agrees to the security environment and either side has the right to close down communication if the requirements are not satisfied.

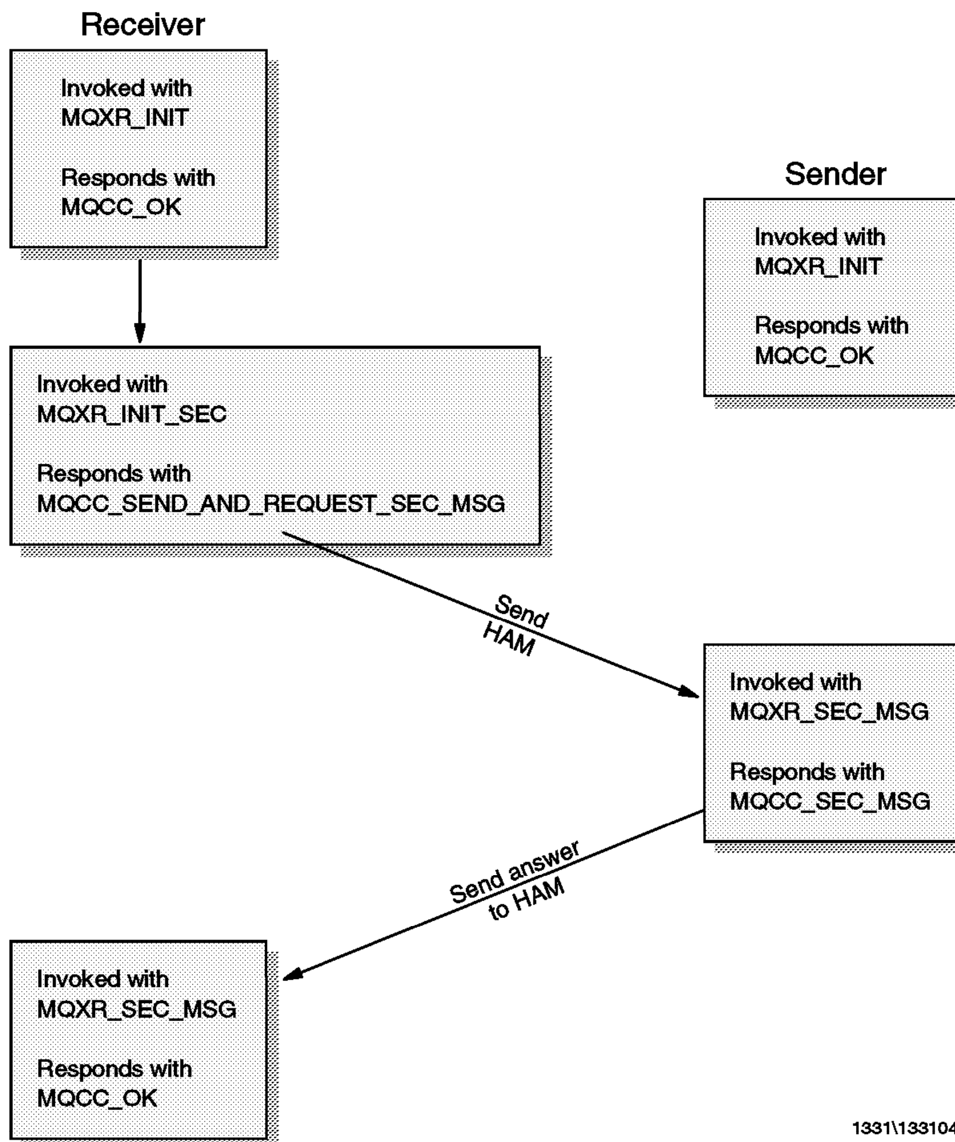


Figure 7. Handshaking Communication Flow

The receiving channel initiates the handshaking procedure. The security exit is invoked by certain events:

1. First, both channel security exits get called with *MQXR_INIT*. Both return *MQXCC_OK* to continue the communication.
2. Next, the receiver's channel security exit gets called with *MQXR_INIT_SEC*. It builds the HAM and initiates the transmission of the HAM to the sender's channel security exit by returning *MQXCC_SEND_AND_REQUEST_SEC_MSG* which also requests a reply to be returned by the sender's channel security exit.
3. The sender's channel security exit gets invoked with *MQXR_SEC_MSG*. It receives the HAM and checks its validity. If the HAM is not valid then the sender's channel security exit returns *MQXX_SUPRESS_FUNCTION* which stops the communication. If the HAM is valid the sender's channel security exit builds a HAM as an answer. The answer must contain the message content of the HAM, just received. It returns *MQXCC_SEND_SEC_MSG* which initiates the transmission of the HAM to the receiver's channel security exit.
4. Finally, the receiver gets invoked with *MQXR_SEC_MSG.* It checks the HAM and if the HAM is valid it returns *MQXCC_OK* and the security environment is established. Now message transmission can begin. If the HAM was not valid *MQXCC_SUPRESS_FUNCTION* will be returned to close the channel.

4.4 Messages

Messages issued by the security exit conform to the following naming convention:

```
MQS9nnns : 9nnns mmmm qqqq MQCHEXIT msgtext
```

This means:

nnn — A 3-digit number identifies the message. The ranges are:

- 1 - 199 for error messages
- 200 - 399 for warnings
- 400 - 599 for informational messages
- 600 - 999 for debugging messages

s — A one letter suffix indicates the message severity.

- D Debugging message
- I Informational message
- W Indicates a warning message

- E Indicates an error message

mmmm The name of the machine where the message came from.

qqqq Queue manager where the message came from.

MQCHEXIT Name of the program which created this message. In our case it's the name of the security exit, MQCHEXIT.

msgtext A meaningful message text that describes what happened.

The naming conventions for messages issued by the channel security exit enable monitoring software to catch the messages easily.

Chapter 5. Channel Security Exit on Different Platforms

This chapter describes:

- How to compile and link the channel security exit.
- How to install the exit on your system.
- How to set up channel definitions for using the exit.
- How to port the application to AIX and OS/390.

5.1 How to Compile and Link a Channel Security Exit on Windows NT

On the Windows NT platform, the channel security exit is a DLL. We used the IBM VisualAge C++ V3.5 compiler to create a project.

To compile MQCHEXIT.C type on the command line:

```
icc  mqchexit /Gm /Ti /Gd /Ge- /Gf- /Fo".\mqchexit.obj" /C mqchexit.c
```

Next, you must create an export definition file MQCHEXIT.EXP using the command:

```
ilib.exe /Gi:mqchexit .\mqchexit.obj
```

Then create the MQCHEXIT.DLL with the following command:

```
icc @mqchexit.inp
```

where mqchexit.inp specifies a response file whose input will be forwarded to the linker.

This file mqchexit.inp looks like this:

```
/B" /de /pmtype:vio /noe /code:RX /data:RW /dll"
/B" /def"
/B" /def:rsaeuro.lib"
/B" /def:wsock32.lib"
/B" /nod:mqchexit.lib"
/Femqchexit.dll
rsaeuro.lib
wsock32.lib
.\mqchexit.exp
.\mqchexit.obj
```

Observe that wsock32.lib is needed for the WIN32 API function gethostname() which provides the machine name and ensures that rsaeuro.lib contains the cryptographic software code.

© Copyright IBM Corp. 1998

5.2 How to Install the Channel Security Exit on Windows NT

The following sections describe what has to be done to install a security exit and how to activate it.

5.2.1 CSSD

CSSD.EXE is used to generate the public and private key files. To do this it needs two input files, located in the same directory where CSSD.EXE is called from:

1. QMGRS.INP
2. CHANNELS.INP

The first file, QMGRS.INP, contains the names of all queue managers where channel security exits will be used. This file is located in the same directory as CSSD.EXE.

Example of QMGRS.INP:

```
WT05219A
WT05219B
```

The second file, CHANNELS.INP, contains all channel names and the names of the queue managers that own them. This file must also reside in the directory from where CSSD.EXE is called.

Example of CHANNELS.INP:

```
WT05219A.TC.WT05219B
WT05219A
WT05219B

PHD.REQUESTQAA
WT05219A
WT05219B
```

The layout of both files is one entry per line. Blank lines are allowed as well as comment lines which are marked with an asterisk in column one.

5.2.2 How to Activate the Channel Security Exit

MQCHEXIT.DLL has to be copied to the directory which is pointed to by the entry *ExitsDefaultPath* in the MQS.INI file.

The directory containing the key files is determined by the value of the environment variable MQSCY_KEYFILEPATH. Make sure to set this variable properly because if the channel security exit isn't able to read these files it

closes down the communication. The private and public key files for that queue manager have to be copied to that directory.

The exit writes messages to a log file whose name is <qmgr-name>.MQCHEXIT.LOG. The location of that log file is determined by the setting of the environment variable MQSCY_LOGFILEPATH. If this variable isn't set properly the exit writes messages to stdout.

5.3 How to Set Up Channel Definitions

In the following sections you find examples on how to define a channel exit for the following channel types:

- Sender and receiver
- Client connection and server connection

5.3.1 How to Define a Sender/Receiver Channel Pair

The following examples show how to define a sender/receiver channel pair to make it ready to use the channel security exit.

5.3.1.1 Sender Channel

```
def channel(WT05219A.TC.WT05219B) chltype(SDR)
    CONNAME('9.24.104.44(1415)') +
    TRPTYPE(tcp) xmitq(WT05219B.TCP.XMITQ) +
    SCYEXIT('MQCHEXIT(ChannelExit)') +
    SENDEXIT('MQCHEXIT(ChannelExit)') +
    SCYDATA('DEBUG') replace
```

The values specified under SCYEXIT() and SENDEXIT() must be exactly as specified in the example above. SCYDATA('DEBUG') is optional. If specified the channel security exit writes debugging messages to the log file.

5.3.1.2 Receiver Channel

```
def channel(WT05219A.TC.WT05219B) chltype(RCVR) +
    SCYEXIT('MQCHEXIT(ChannelExit)')
    RCVEXIT('MQCHEXIT(ChannelExit)')
```

In this example, SCYDATA() isn't specified and the channel security exit writes no debugging messages to the log file. As before, SCYEXIT() and RCVEXIT() entries are required exactly as specified in the example.

If the appropriate definitions are done for the participating queue managers the channel security exit gets invoked by the MCA (Message Channel Agent).

5.3.2 How to Define a SVRCONN/CLNTCONN Channel Pair

For a client application, there has to exist a svrconn channel on the server′s side and a clntconn channel on the client′s side. The following examples shows how to define a svrconn/clntconn channel pair to make it ready to use the channel security exit.

5.3.2.1 Server Connection Channel

```
def channel(WT05219A.SVRCONN) +
    chltype(SVRCONN) TRPTYPE(TCP) +
    SCYEXIT(′MQCHEXIT(ChannelExit)′) +
    RCVEXIT(′MQCHEXIT(ChannelExit)′)
```

Here also the entries for SCYEXIT() and RCVEXIT() have to be defined exactly as shown.

5.3.2.2 Client Connection Channel

```
def channel(WT05219A.SVRCONN) chltype(CLNTCONN) +
    CONNAME(′10.34.56.23(1414)′) TRPTYPE(TCP)  +
    SCYEXIT(′MQCHEXIT(ChannelExit)′) +
    RCVEXIT(′MQCHEXIT(ChannelExit)′) +
    SENDDATA(WTPING)
```

As before, the entries for SCYDATA() and RCVEXIT() have to be as shown.

The value specified in SENDDATA is taken by the channel security exit to find its key files. In our example, the names of the key files would be: WTPING.PRV and WTPING.PUB. This definition is stored in a channel definition table and delivered to the customer together with the application.

5.4 How to Port the Exit to AIX

Porting the channel security exit to AIX is straightforward. Instead of writing to a log file, messages should be written to the system log.

- The function openlog() initializes writing to the AIX syslog. Example:

```
#include <syslog.h>
...
  openlog( ″MQSECEXIT″, LOG_PID, LOG_USER );
```

- The function syslog() writes to the syslog. Example:

```
#include <syslog.h>
...
char *errstr;
...
syslog( LOG_INFO, errstr );
```

- The WIN32 API function gethostname() is also available in an AIX environment. Example:

```
char            hostname[256];
...
gethostname( hostname, 255 );
```

5.5 How to Port the Exit to OS/390

There are some more points to consider when porting the application to OS/390. Messages should be written to the console.

A sample is available from CBC.SCBCSAM. Module CBC3GWT1 is an assembler module to do the WTO:

```
     6,=A(ACTMSG)        SET SVC35.ACTMSG TO DYN MSG
         LA    7,76                LEN(WTO MESSAGE)-SET MAX 76
         L     5,0(,1)             PARM1 IS LENGTH OF DYN MSG
         L     5,0(,5)
         O     5,=X'40000000'      1ST BYTE - PAD CHAR (' ')
         L     4,4(,1)             PARM2 IS DYN MSG ADDR
         MVCL  6,4                 COPY DYNMSG TO SVC35 STRUCT
         CNOP  0,4
         BAL   1,BARNDMSG          BRANCH AROUND SVC35 STRUCT
         DC    AL2(80)             TEXT LENGTH (76+4)
         DC    B'1000000000000000' MCSFLAGS
ACTMSG   DC    CL76' '             ARBITRARY SIZE OF 76
         DC    B'0000000000000000' DESCRIPTOR CODES
         DC    B'0100000000000000' ROUTING CODES
BARNDMSG DS    0H
         SVC   35                  ISSUE SVC 35
         EDCEPIL
         END
```

Module CBC3GWT2 is a C program to show how to call dynwto:

```
/* write to operator example */
/* part 2 of 2-other file is CBC3GWT1 */
   #pragma linkage(dynwto,OS)
   void dynwto(int, char *);
   main()
   {
    dynwto(9,"something");
   }
```

There is a function __iphost() to retrieve the resolver-supplied hostname. If the keyword HOSTNAME isn't found in the resolver configuration data a null

string is returned; otherwise the function returns the hostname of the machine the function is running on.

File I/O is a bit different because we have to use DD-Names.

Another important point is to consider the byte ordering of the underlying hardware. The S/390 is a big endian machine; that is, bytes are processed the same way they are stored. Intel architecture is a little endian machine.

Because we did not check out if RSAEURO supports "big endian" architecture it could be a problem to port it to the OS/390 platform.

Here is an example of how a function can detect at runtime if it is running on a big endian or little endian machine.

```
 IsLittleEndian()
{
   unsigned short   s = 1;

  /* If little endian storage representation of s is: 0x01,0x00.
   * If big endian storage representation of s is: 0x00, 0x01
   * For example S/370 is big endian and Intel is little endian.
   */

   if ( *(char *)&s == 0)
      return 0;
   else
      return 1;
}
```

Appendix A. MQSeries Channel Security Exit Source Code

This appendix includes the header files and source code for the C application using on the AIX and Windows NT systems described in this document.

- A.1, “Header File MQCHEXIT.H”
- A.2, “Header File MQSEC.H” on page 34
- A.3, “Source Code of CSSD.C” on page 37
- A.4, “Source Code of MQCHEXIT.C” on page 47

These files are also on the diskette that comes with the book.

A.1 Header File MQCHEXIT.H

```
/*-------------------------------------------------------
* This file was created for Manfred Lotz on 05/11/98.
* VisualAge for C++ Version 3.5, (C) Copyright IBM Corp. 1996
*------------------------------------------------------*/

#pragma library("mqchexit.lib")

#include <stddef.h>

#include "mqsec.h"

extern void MQENTRY   ChannelExit( PMQVOID channelExitParms,
                                   PMQVOID channelDef,
                                   PMQLONG dataLength,
                                   PMQLONG agBufLength,
                                   PMQVOID agBuf,
                                   PMQLONG exitBufLength,
                                   PMQPTR exitBufAddr );
```

Figure 8. MQCHEXIT.H

© Copyright IBM Corp. 1998

A.2 Header File MQSEC.H

```
#ifndef _MQSEC_H_
#define _MQSEC_H_

#include "cmqc.h"
#include "rsaeuro.h"            /* include rsaeuro function header file */
#define     TRUE                1
#define     FALSE               0

typedef   signed short        SHORT;
typedef   unsigned char       UBYTE;
typedef   unsigned short      USHORT;
typedef   unsigned long       ULONG;

#ifndef _WIN32
typedef   unsigned char       BOOL;
#endif

#ifdef  MLDEBUG
# define D(x) x                 /* expand only when debugging     */
# define ND(x)                  /* expand only when not debugging */
#else
# define D(x)
# define ND(x) x
#endif

#define  MQRE_HAM_MSGLEN  0x0901
#define  MQRE_HAM_MSG  0x0902

typedef struct
{
 unsigned int  errorno;
 char          *errmsg;

} MQRE_MSG;
```

Figure 9 (Part 1 of 3). MQSEC.H

```
typedef struct
{
    unsigned char   iv[8];
    unsigned char   encryptedKey[MAX_ENCRYPTED_KEY_LEN];
    unsigned int    encryptedKeyLen;

    unsigned char   signature[MAX_SIGNATURE_LEN];
    unsigned int    signatureLen;

}   MQ_DIGITAL_SIGNATURE;

typedef struct
{
    unsigned char   hambuf[256];
    unsigned int    hambuflen;
    unsigned char   hamEncryptedBuf[32];
    unsigned int    hamEncryptedBufLen;
    MQ_DIGITAL_SIGNATURE mqdigsig;
}   MQ_HAM;

typedef struct
{
 unsigned char qmgr[MQ_Q_MGR_NAME_LENGTH+1];
 unsigned int  qmgrLen;
 R_RSA_PUBLIC_KEY   pubk;
 R_RSA_PRIVATE_KEY  privk;
}   MQ_QMGR_DATA;


typedef struct
{
 unsigned char   channel[MQ_CHANNEL_NAME_LENGTH+1];
 unsigned char   qmgrA[MQ_Q_MGR_NAME_LENGTH+1];
 unsigned char   qmgrB[MQ_Q_MGR_NAME_LENGTH+1];
    MQ_DIGITAL_SIGNATURE mqdigsig;
}   MQ_CHANNEL_DATA;
```

Figure 9 (Part 2 of 3). MQSEC.H

```
/* Define eyecatchers */

#define     QMGR_NAME_EYE_CATCHER          "<QUEUEMANAGER NAME>"
#define     QMGR_NAME_EYE_CATCHER_SIZE     sizeof(QMGR_NAME_EYE_CATCHER)

#define     CSSD_PUBK_EYE_CATCHER          "<CSSD PUBLIC KEY>"
#define     CSSD_PUBK_EYE_CATCHER_SIZE     sizeof(CSSD_PUBK_EYE_CATCHER)

#define     QMGR_PUBK_EYE_CATCHER          "<QMGR PUBLIC KEY>"
#define     QMGR_PUBK_EYE_CATCHER_SIZE     sizeof(QMGR_PUBK_EYE_CATCHER)

#define     QMGR_PRIVK_EYE_CATCHER         "<QMGR PRIVATE KEY>"
#define     QMGR_PRIVK_EYE_CATCHER_SIZE    sizeof(QMGR_PRIVK_EYE_CATCHER)

#define     CHANNEL_NAME_EYE_CATCHER       "<CHANNEL NAME>"
#define     CHANNEL_NAME_EYE_CATCHER_SIZE  sizeof(CHANNEL_NAME_EYE_CATCHER)

#define     DIGSIG_EYE_CATCHER             "<DIGITAL SIGNATURE>"
#define     DIGSIG_EYE_CATCHER_SIZE        sizeof(DIGSIG_EYE_CATCHER)


#define     HAM_EYE_CATCHER                "<HAM>"
#define     HAM_EYE_CATCHER_SIZE           sizeof(HAM_EYE_CATCHER)

#define     MSG_EYE_CATCHER                "<MESSAGE>"
#define     MSG_EYE_CATCHER_SIZE           sizeof(MSG_EYE_CATCHER)

#define     MQ_DO_COMPRESS         1
#define     MQ_DO_NO_COMPRESS      0
#define     MQ_FLAG_Copied         (UBYTE ) 0x80
#define     MQ_FLAG_Compress       (UBYTE) 0x40

#endif
```

Figure 9 (Part 3 of 3). MQSEC.H

A.3 Source Code of CSSD.C

```
#include <stdio.h>
#include <stdlib.h>
#include "rsaeuro.h"
#include "mqsec.h"

void    GetDigest( unsigned char  *digestOut,
                   unsigned int   *digestLen,
                   unsigned char  *buf,
                   unsigned int   bufLen );
int     CreateKeyPair( R_RSA_PUBLIC_KEY  * pubk,
                       R_RSA_PRIVATE_KEY * privk,
                       unsigned char     *buf,
                       unsigned int      buflen);
// unsigned char      buffer[4096];
MQ_QMGR_DATA       mqQmgrData[80];
unsigned int       noOfQmgrs;
MQ_CHANNEL_DATA    mqChannelData[1024];
unsigned int       noOfChannels;
R_RSA_PUBLIC_KEY   *cssdPubk;
R_RSA_PRIVATE_KEY  *cssdPrivk;

static void InitQmgrFromFile(void);
static void InitChannelsFromFile(void);
static void SetKeys( MQ_QMGR_DATA *qmgrsec );
static void DisplayQmgrs( MQ_QMGR_DATA *qmgrsec );
static void DisplayChannels( MQ_CHANNEL_DATA *channeldata );
static void CreateFiles(MQ_QMGR_DATA *qmgrsec);
static void WritePrivateFile(unsigned char *fname, MQ_QMGR_DATA *qmgrsec);
static void WritePublicFile(unsigned char *fname, MQ_QMGR_DATA *qmgrsec);
static
    unsigned char* GetRecord(unsigned char *buf,unsigned int maxlen,FILE *fptptr);
static MQ_QMGR_DATA *GetQmgrData(unsigned char *qmgr);
static void PrintBuff (  unsigned char *out, int len ) ;

int main(int argc, char **argv, char **envp)
{
    InitQmgrFromFile();
    InitChannelsFromFile();
    CreateFiles(mqQmgrData);
    return 0;
}
```

Figure 10 (Part 1 of 10). CSSD.C

```
static void InitQmgrFromFile(void)
{
 FILE           *fPtr;
 MQ_QMGR_DATA *qmgrPtr = mqQmgrData+1;
/* Initialize with 1 because we do CSSD outside the loop */
 unsigned int  i = 1;
    unsigned char buf[256];
 unsigned char   *bufPtr;

 fPtr = MQ_OpenFile( "QMGRS.INP" ,"r");

 if ( fPtr == NULL )
 {
  printf( "Cannot open QMGRS.INP\n");
  exit(8);
 }

    strcpy(mqQmgrData[0].qmgr,"CSSD");
    SetKeys(mqQmgrData);

 while (GetRecord( buf, 256, fPtr ) )
 {
       strcpy( qmgrPtr->qmgr,buf );
       // if ( qmgrPtr->qmgr[0] == ' ' ]] qmgrPtr->qmgr[0] == '*' )
         //   continue;
  SetKeys(qmgrPtr);
       // PrintBuff( (unsigned char*) qmgrPtr->qmgr, MQ_Q_MGR_NAME_LENGTH );

  qmgrPtr += 1; /* point to next element in array */

  /* Make sure we won't overrun the array */
  if ( i++ == 80 )
   break;
 }

 noOfQmgrs = i;
 MQ_CloseFile( fPtr );

    cssdPubk        =   & mqQmgrData[0].pubk;
    cssdPrivk       =   & mqQmgrData[0].privk;
 /* Only for debugging */
 // DisplayQmgrs(mqQmgrData);
}
```

Figure 10 (Part 2 of 10). CSSD.C

```
static void InitChannelsFromFile(void)
{
 unsigned char buf[256];
 unsigned char   *bufPtr;
 FILE            *fPtr;
 MQ_CHANNEL_DATA  *channelPtr = mqChannelData;
 unsigned int  i = 0;
 unsigned char *token;
// unsigned char   seps[] = " ";


 fPtr = MQ_OpenFile( "CHANNELS.INP" ,"r");

 if ( fPtr == NULL )
 {
  perror( "Cannot open CHANNELS.INP\n");
  exit(8);
 }

 while ( GetRecord( buf, 256, fPtr ) )
 {
        strcpy(channelPtr->channel,buf);
        GetRecord( buf, 256, fPtr );
        strcpy(channelPtr->qmgrA,buf);
        GetRecord( buf, 256, fPtr );
        strcpy(channelPtr->qmgrB,buf);

       // printf("Channel %s, qmgrA %s, qmgrB %s\n",
            channelPtr->channel, channelPtr->qmgrA, channelPtr->qmgrB);
  channelPtr += 1; /* point to next element in array */

  /* Make sure we won't overrun the array */
  if ( i++ == 1024 )
   break;
 }

 noOfChannels = i;

 MQ_CloseFile( fPtr );

 /* Only for debugging */
 DisplayChannels(mqChannelData);

}
```
Figure 10 (Part 3 of 10). CSSD.C

```
static void SetKeys( MQ_QMGR_DATA *qmgrsec )
{
 unsigned int   status;

 /* Initialize the struct */
 qmgrsec->qmgrLen = strlen(qmgrsec->qmgr)+1;

 /* Get rid of the newline character in the string */
 qmgrsec->qmgrLen -= 1;
 qmgrsec->qmgr[qmgrsec->qmgrLen ] = 0x00;
    printf("Qmgrname: %s Length: %d\n",qmgrsec->qmgr, qmgrsec->qmgrLen);
 status = CreateKeyPair( &( qmgrsec->pubk ),
          &( qmgrsec->privk ),
          qmgrsec->qmgr,
          qmgrsec->qmgrLen);
    if ( status )
        printf("Key generation failed with return code: %02lX\n",status);
}


static void DisplayQmgrs( MQ_QMGR_DATA *qmgrsec )
{

 printf("%d queue manager names read from file\n",noOfQmgrs);
 while ( qmgrsec->qmgr[0] != 0 )
 {
  printf("%-14s %d \n", qmgrsec->qmgr, qmgrsec->qmgrLen );
  qmgrsec += 1;
 }
}


static void DisplayChannels( MQ_CHANNEL_DATA *channeldata )
{

      printf("%d channel names read from file\n",noOfChannels);
      while ( channeldata->channel[0] != 0 )
      {
            printf("%-20s %14s %14s \n", channeldata->channel,
                   channeldata->qmgrA, channeldata->qmgrB );
            channeldata += 1;
      }
}
```

Figure 10 (Part 4 of 10). CSSD.C

```
int             CreateKeyPair( R_RSA_PUBLIC_KEY * pubk,
                               R_RSA_PRIVATE_KEY * privk,
                               unsigned char *buf,
                               unsigned int buflen)
{
    R_RANDOM_STRUCT random;
    R_RSA_PROTO_KEY protoKey;
    int             status;
    unsigned char   digestOut[MAX_DIGEST_LEN];
    unsigned int    digestLen;


    R_RandomInit( &random );

    /* Key size in bits */
    protoKey.bits = 512;

    protoKey.useFermat4 = 1;

    GetDigest(digestOut, &digestLen, buf, buflen);

    status = R_RandomUpdate(&random, digestOut, digestLen );
    R_RandomFinal(&random);


    //R_RandomCreate(&random);
    status = R_GeneratePEMKeys( pubk, privk, &protoKey, &random );

    printf("R_GeneratePEMKeys() ==> %d\n",status);

    // PrintBuff( (unsigned char*) pubk, sizeof(R_RSA_PUBLIC_KEY) );
    // PrintBuff( (unsigned char*) privk, sizeof(R_RSA_PRIVATE_KEY) );

    return status;
}
```

Figure 10 (Part 5 of 10). CSSD.C

```
/*
 * Generating a message digest of memory-resident data
 */
 void GetDigest(unsigned char  *digestOut,
                unsigned int   *digestLen,
                unsigned char  *buf,
                unsigned int   bufLen )
 {
  R_DIGEST_CTX md5ctxt;
  unsigned int  i;


  R_DigestBlock(digestOut, digestLen, buf, bufLen, DA_MD5);
 }



 static void CreateFiles(MQ_QMGR_DATA *qmgrsec)
 {
 unsigned char   privFileName[256];
    unsigned char   publicFileName[256];
 unsigned int  i;

 for ( i = 1; i<noOfQmgrs; i++ )
 {
   /* Build file private file name */
   strcpy(privFileName,qmgrsec[i].qmgr);
   strcat(privFileName,".PRV");
   WritePrivateFile(privFileName, &(qmgrsec[i]));

            strcpy(publicFileName,qmgrsec[i].qmgr);
   strcat(publicFileName,".PUB");
            WritePublicFile(publicFileName, &(qmgrsec[i]));
 }
}
```

Figure 10 (Part 6 of 10). CSSD.C

```
static void WritePrivateFile(unsigned char *fname, MQ_QMGR_DATA *qmgrsec)
{
 FILE      *fptr;
 unsigned int    len;
 unsigned char   digestOut[MAX_DIGEST_LEN];
 unsigned int   digestLen;

 printf("Writing private file %s\n", fname);
 if ( (fptr = MQ_OpenFile(fname,"wb")) == NULL )
 {
  printf("Cannot open file %s\n",fname);
  exit(16);
 }

 /* We always prepend data with an appropiate eye catcher */

 /* Write queue manager's name to file */
    fwrite( QMGR_NAME_EYE_CATCHER, QMGR_NAME_EYE_CATCHER_SIZE, 1, fptr);
 fwrite( qmgrsec->qmgr, MQ_Q_MGR_NAME_LENGTH, 1, fptr );

 /* Write the CSSD's public key to file */
    fwrite( CSSD_PUBK_EYE_CATCHER, CSSD_PUBK_EYE_CATCHER_SIZE, 1, fptr);
 fwrite( &( mqQmgrData[0].pubk ), sizeof(R_RSA_PUBLIC_KEY), 1, fptr );

 /* Write the queue manager's public key to file */
    fwrite( QMGR_PUBK_EYE_CATCHER, QMGR_PUBK_EYE_CATCHER_SIZE, 1, fptr);
 fwrite( &( qmgrsec->pubk ), sizeof(R_RSA_PUBLIC_KEY), 1, fptr );

 /* Write the queue manager's private key to file */
    fwrite( QMGR_PRIVK_EYE_CATCHER, QMGR_PRIVK_EYE_CATCHER_SIZE, 1, fptr);
 fwrite( &( qmgrsec->privk ), sizeof(R_RSA_PRIVATE_KEY), 1, fptr );

 MQ_CloseFile(fptr);
}
```

Figure 10 (Part 7 of 10). CSSD.C

```
static void WritePublicFile(unsigned char *fname, MQ_QMGR_DATA *qmgrsec)
{
    FILE              *fptr;
    unsigned int      len;
    unsigned char     signature[MAX_SIGNATURE_LEN];
    unsigned int      signatureLen;
    unsigned int      i;
    unsigned char     *qmgrPtr;
    MQ_QMGR_DATA      *qmgrdata;
    R_RSA_PUBLIC_KEY  *pubk;
    unsigned char     buf[1024];
    int               status;
    R_SIGNATURE_CTX   context;

 printf("Writing public file %s\n", fname);
 if ( (fptr = MQ_OpenFile(fname,"wb")) == NULL )
 {
  printf("Cannot open file %s\n",fname);
  exit(16);
 }
    for ( i=0; i<noOfChannels; i++ )
    {
        printf("A: %s B: %s  Q: %s\n",mqChannelData[i].qmgrA,
               mqChannelData[i].qmgrB, qmgrsec->qmgr);
        if ( strcmp(mqChannelData[i].qmgrA,qmgrsec->qmgr) == 0 )
        {
            qmgrdata = GetQmgrData(mqChannelData[i].qmgrB );
            if ( qmgrdata == NULL )
            {
                printf("Couldn't find public key of qmgr %s\n",
                       mqChannelData[i].qmgrB);
                return;
            }
        }
        else if ( strcmp(mqChannelData[i].qmgrB,qmgrsec->qmgr) == 0 )
        {
            qmgrdata = GetQmgrData(mqChannelData[i].qmgrA );
            if ( qmgrdata == NULL )
            {
             printf("Couldn't find public key of qmgr %s\n",mqChannelData[i].qmgrA);
             return;
            }
        }
```

Figure 10 (Part 8 of 10). CSSD.C

```
        else continue;;
        /* Write channel name */
        fwrite( CHANNEL_NAME_EYE_CATCHER, CHANNEL_NAME_EYE_CATCHER_SIZE, 1, fptr
        fwrite( mqChannelData[i].channel, MQ_CHANNEL_NAME_LENGTH, 1, fptr);

        /* Write partner qmgr name */
        fwrite( QMGR_NAME_EYE_CATCHER, QMGR_NAME_EYE_CATCHER_SIZE, 1, fptr);
        fwrite( qmgrdata->qmgr, MQ_Q_MGR_NAME_LENGTH, 1, fptr );

        /* Write partner qmgr's public key */
        fwrite( QMGR_PUBK_EYE_CATCHER, QMGR_PUBK_EYE_CATCHER_SIZE, 1, fptr);
     fwrite( &qmgrdata->pubk, sizeof(R_RSA_PUBLIC_KEY), 1, fptr);

        /* Create and write a digital signature */
        status = R_SignInit(&context, DA_MD5);

        status = R_SignUpdate(&context, mqChannelData[i].channel,
                              MQ_CHANNEL_NAME_LENGTH);
        status = R_SignUpdate(&context, qmgrdata->qmgr, MQ_Q_MGR_NAME_LENGTH);
        status = R_SignUpdate(&context, (unsigned char *)
                              &qmgrdata->pubk, sizeof(R_RSA_PUBLIC_KEY));
        status = R_SignFinal(&context, signature, &signatureLen, cssdPrivk);

        fwrite( DIGSIG_EYE_CATCHER, DIGSIG_EYE_CATCHER_SIZE, 1, fptr);
        fwrite( &signatureLen, sizeof(unsigned int), 1, fptr);
        fwrite( signature, MAX_SIGNATURE_LEN, 1, fptr);
    }

 MQ_CloseFile(fptr);
}

/* Returns a pointer to the QMGR_DATA structure */
MQ_QMGR_DATA *GetQmgrData(unsigned char *qmgr)
{
    unsigned int        i;

    /* We begin with i=1 because we skip the CSSD */
    for ( i=1; i<noOfQmgrs; i++ )
    {
        if (strcmp(qmgr,mqQmgrData[i].qmgr) == 0 )
            return mqQmgrData+i;
    }
    return 0;
}
```

Figure 10 (Part 9 of 10). CSSD.C

```
static unsigned char* GetRecord(unsigned char *buf,unsigned int maxlen,FILE *fptr)
{
    unsigned char *bufPtr;

    while ( 1 )
    {
        if ( (bufPtr = fgets(buf,maxlen,fptr)) == 0 )
            return bufPtr;

        if ( buf[0] == ' ' ]] buf[0] == '*' ]] buf[0] == '\n')
            continue;

        /* Remove trailing \n */
        buf[strlen(buf)-1] = 0x00;

        /* Removing trailing blanks */
        while ( *buf != ' ' )
        {
            if ( *buf == 0x00 )
                break;
            buf += 1;
        }
        *buf = 0x00;
        return bufPtr;
    }


}

static void     PrintBuff( unsigned char *out, int len )
{
    int             i;

    for ( i = 0; i < len; i++ )
        printf( "%02X", *( out + i ) );

    printf( "\n" );
}
```

Figure 10 (Part 10 of 10). CSSD.C

A.4 Source Code of MQCHEXIT.C

```
/*----------------------------------------------------------
* This file was created for Manfred Lotz on 05/11/98.
* VisualAge for C++ Version 3.5, (C) Copyright IBM Corp. 1996
*---------------------------------------------------------*/

/*---------------------------------------------
 * mrchexit.c - Source file for a C DLL
 *---------------------------------------------*/
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>

#include <cmqc.h>
#include <cmqxc.h>

#include <windows.h>
#include <winsock.h>

#include "mqchexit.h"

#ifdef _AIX
#include <syslog.h>
#endif

/* Names of environmentvariables used to locate key files and log files */
#define KEYFILE_PATH_VAR        "MQSCY_KEYFILEPATH"
#define LOGFILE_PATH_VAR        "MQSCY_LOGFILEPATH"

/* Trap messages. */
#define NORMAL_TRAP 1100
#define CRITICAL_TRAP 1200

#define MQS_HAM_BUFLEN   1024
```

Figure 11 (Part 1 of 42). MQCHEXIT.C

```
static FILE        *logFilePtr        = 0;
static char        logFileName[256];
static char        logFilePath[256];


static  int        doDebugging = 0;


static char        keyFilePath[256];
unsigned char      privKeyFileName[256];
unsigned char      pubKeyFileName[256];



static char        hostName[256];
unsigned int       hostNameLen;




MQ_QMGR_DATA       qmgrdata;
MQ_CHANNEL_DATA    channeldata;
R_RSA_PUBLIC_KEY  cssdPubk;
R_RSA_PUBLIC_KEY  partnerpubk;
MQCHAR             partnername[MQ_CHANNEL_NAME_LENGTH+1];



MQ_HAM          origMqhamMsg;
MQ_HAM          mqhamMsg;

unsigned char   tmpBuf[1024];
```

Figure 11 (Part 2 of 42). MQCHEXIT.C

```
/* Prototypes */

void _Export MQENTRY ChannelExit( PMQVOID channelExitParms,
                                  PMQVOID channelDef,
                                  PMQLONG dataLength,
                                  PMQLONG agBufLength,
                                  PMQVOID agBuf,
                                  PMQLONG exitBufLength,
                                  PMQPTR exitBufAddr );






void MQENTRY    SecurityChannelExit( PMQVOID channelExitParms,
                                     PMQVOID channelDef,
                                     PMQLONG dataLength,
                                     PMQLONG agBufLength,
                                     PMQVOID agBuf,
                                     PMQLONG exitBufLength,
                                     PMQPTR exitBufAddr );

void MQENTRY    SendMessageExit( PMQVOID channelExitParms,
                                 PMQVOID channelDef,
                                 PMQLONG dataLength,
                                 PMQLONG agBufLength,
                                 PMQVOID agBuf,
                                 PMQLONG exitBufLength,
                                 PMQPTR exitBufAddr );

void MQENTRY    ReceiveMessageExit( PMQVOID channelExitParms,
                                    PMQVOID channelDef,
                                    PMQLONG dataLength,
                                    PMQLONG agBufLength,
                                    PMQVOID agBuf,
                                    PMQLONG exitBufLength,
                                    PMQPTR exitBufAddr );




void            MQStart( void );
```

Figure 11 (Part 3 of 42). MQCHEXIT.C

```
int             ReadKeysFromFiles(void);
int             ReadPrivateKey(void);
int             ReadPublicKey(void);

unsigned char   *AllocExitBuffer( int len );
void            FreeExitBuffer(PMQPTR exitBufAddr, PMQLONG exitBufLength );
void            MakeCString( char *bf, char *zt, int len );
void            GetInfo( PMQCD pChDef);
int             GetHostName(unsigned char *buf, unsigned int *bufLen);
void            BuildLogFileName(void);


void            WriteLogFile( int msgid, unsigned char msgtype,
unsigned char *msgtxt,...);
void            PrintBuff( unsigned char *out, unsigned int len );




int             BuildHAM( char *buf , unsigned int *bufLen, MQ_HAM *mqham);
int             ReceiveHAM( MQ_HAM *mqham, char *buf, unsigned int buflen );
int            ReadAnswerHAM( MQ_HAM *mqham, char *buf, unsigned int buflen);

int             BuildBufferToSendHAM( unsigned char *outbuf,
                                    unsigned int *outbuflen,
                                         MQ_HAM * mqham );

int             EncryptHAM( MQ_HAM               *mqham,
                   R_RSA_PUBLIC_KEY     *pubk,
                   R_RSA_PRIVATE_KEY    *privk );

int             InitHAM( MQ_HAM * mqham );


int             EncryptBuffer( unsigned char *outbuf,
                                  unsigned int *outbuflen,
                                  unsigned char *inbuf,
                                  unsigned int inbuflen,
                                  unsigned int compressflag,
                                  R_RSA_PUBLIC_KEY * recpubk,
                               R_RSA_PRIVATE_KEY * sdrprivk,
                          MQ_DIGITAL_SIGNATURE * mqdigsig );
```

Figure 11 (Part 4 of 42). MQCHEXIT.C

```
int             DecryptAndVerifyBuffer( unsigned char *outbuf,
                                         unsigned int *outbuflen,
                                         unsigned char *inbuf,
                                         unsigned int inbuflen,
                                         R_RSA_PUBLIC_KEY * sdrpubk,
                                       R_RSA_PRIVATE_KEY * recprivk,
                                MQ_DIGITAL_SIGNATURE * mqdigsig );



int             FillRandomBuffer( unsigned char *outbuf, int len );

int             FillRandomBuffer16( unsigned char *outbuf );


int             ReadHAM( MQ_HAM * mqham,
                               unsigned char *rcvrbuf,
                               unsigned int rcvrbuflen );

int             VerifyHAM( MQ_HAM * mqham,
                              R_RSA_PUBLIC_KEY * pubk,
                              R_RSA_PRIVATE_KEY * privk );


int             VerifyAnswerHAM( MQ_HAM * mqham, MQ_HAM *origmqham,
                                      R_RSA_PUBLIC_KEY * sdrpubk,
                                 R_RSA_PRIVATE_KEY * rcvrprivk );


unsigned char*  BuildBufferToSendMSG( unsigned int *outbuflen,
                                        int          *status,
                                     unsigned char *inbuf,
                                     unsigned int inbuflen );


unsigned char*  ReceiveMSGBuffer( unsigned int *outbuflen,
                                     unsigned char *inbuf,
                                     int       *status,
                                     unsigned int inbuflen );
```

Figure 11 (Part 5 of 42). MQCHEXIT.C

```
/* This is the actual ChannelExit routine which is exported and
 * will be invoked from MQSeries.
 * It acts as a dispatcher to call the approriate exit routines:
 *      SecurityChannelExit()
 *      SendMessageExit()
 *      ReceiveMessageExit()
 *
 */
void _Export MQENTRY ChannelExit( PMQVOID channelExitParms,
                                                  PMQVOID channelDef,
                                                  PMQLONG dataLength,
                                                  PMQLONG agBufLength,
                                                  PMQVOID agBuf,
                                                  PMQLONG exitBufLength,
                                                  PMQPTR exitBufAddr )

{
    PMQCXP          pChlExParms = ( PMQCXP ) channelExitParms;
    PMQCD           pChDef = ( PMQCD ) channelDef;
    static int      firstTime          = 1;





    if ( firstTime  )
    {
        GetInfo(pChDef);
        BuildLogFileName();
        firstTime   = 0;
    }

    switch ( pChlExParms->ExitId )
    {
```

Figure 11 (Part 6 of 42). MQCHEXIT.C

```
        case MQXT_CHANNEL_SEC_EXIT:
            WriteLogFile( 400, 'I', ">>>>> Security exit started\n");
            SecurityChannelExit( channelExitParms,
                                 channelDef,
                                 dataLength,
                                 agBufLength,
                                 agBuf,
                                 exitBufLength,
                                 exitBufAddr );
            break;
        case MQXT_CHANNEL_SEND_EXIT:
            WriteLogFile( 410, 'I', ">>>>> Send message exit started\n");
            SendMessageExit( channelExitParms,
                             channelDef,
                             dataLength,
                             agBufLength,
                             agBuf,
                             exitBufLength,
                             exitBufAddr );
            break;
        case MQXT_CHANNEL_RCV_EXIT:
            WriteLogFile( 420, 'I', ">>>>> Receive message exit started\n" );
            ReceiveMessageExit( channelExitParms,
                                channelDef,
                                dataLength,
                                agBufLength,
                                agBuf,
                                exitBufLength,
                                exitBufAddr );
            break;
        case MQXT_CHANNEL_MSG_EXIT:
             WriteLogFile( 1, 'E',
                           "Channel message exit will not be handled here\n");
        default:;
    }                           /* endswitch ExitType */


}
```

Figure 11 (Part 7 of 42). MQCHEXIT.C

```
/*
 * This is the function for the channel security exit
 */
void MQENTRY    SecurityChannelExit( PMQVOID channelExitParms,
                                      PMQVOID channelDef,
                                      PMQLONG dataLength,
                                      PMQLONG agBufLength,
                                      PMQVOID agBuf,
                                      PMQLONG exitBufLength,
                                      PMQPTR exitBufAddr )

{
    int             i = 0;
    unsigned long   ExitBufferLen = 0;
    char           *pExitBuffer = ( char * ) agBuf;
    char           *pagBuf = ( char * ) agBuf;
    PMQCXP          pChlExParms = ( PMQCXP ) channelExitParms;
    PMQCD           pChDef = ( PMQCD ) channelDef;
    time_t          datetime;
    int             status;
    char            *buf;
    unsigned int    bufLen;

    MQStart(  );
    FreeExitBuffer( exitBufAddr, exitBufLength );


    switch ( pChlExParms->ExitReason )
    {
        case MQXR_INIT:
            WriteLogFile( 600, 'D', "ExitReason = MQXR_INIT\n" );

            status = ReadKeysFromFiles();
            if ( status != 0 )
            {
                WriteLogFile( 12, 'E',
                "Reading keys from file unsuccessful. Error code: 0x%04lX\n",
                status );
                pChlExParms->ExitResponse = MQXCC_SUPPRESS_FUNCTION;
                pChlExParms->Feedback = 0L;
                pChlExParms->ExitResponse2 = 0L;
                break;
            }
```

Figure 11 (Part 8 of 42). MQCHEXIT.C

```
            pChlExParms->ExitResponse2 = 0L;
            pChlExParms->Feedback = 0L;
            pChlExParms->ExitResponse = MQXCC_OK;
            WriteLogFile( 605, 'D', "return(MQXCC_OK)\n" );
            break;

        case MQXR_TERM:
            WriteLogFile( 610, 'D', "ExitReason = MQXR_TERM\n" );
            pChlExParms->ExitResponse = MQXCC_OK;
            WriteLogFile( 605, 'D', "return(MQXCC_OK)\n" );
            break;
        case MQXR_MSG:
            WriteLogFile( 611, 'D', "ExitReason = MQXR_MSG\n" );
            WriteLogFile( 605, 'D', "return(MQXCC_OK)\n" );
            break;
        case MQXR_XMIT:
            WriteLogFile( 612, 'D', "ExitReason = MQXR_XMIT\n" );
            WriteLogFile( 605, 'D', "return(MQXCC_OK)\n" );
            break;
        case MQXR_INIT_SEC:
            WriteLogFile( 620, 'D', "ExitReason = MQXR_INIT_SEC\n" );
            switch ( pChDef->ChannelType )
            {
                case MQCHT_SENDER:
                WriteLogFile( 621, 'D', "Channel type = MQCHT_SENDER\n" );
                WriteLogFile( 606, 'D', "return(MQXCC_SEND_SEC_MSG)\n" );
                    break;

                case MQCHT_SVRCONN:
                WriteLogFile( 626, 'D', "Channel type = MQCHT_SVRCONN\n" );
                case MQCHT_RECEIVER:
                WriteLogFile( 622, 'D', "Channel type = MQCHT_RECEIVER\n" );

                    buf = AllocExitBuffer( MQS_HAM_BUFLEN );
                    if ( buf == NULL )
                    {
                        WriteLogFile( 56, 'E',
                          "Buffer allocation failed for ExitBuffer\n" );
                        pChlExParms->ExitResponse = MQXCC_SUPPRESS_FUNCTION;
                        pChlExParms->ExitResponse2 = MQXR2_USE_AGENT_BUFFER;
                    }
                    else
```

Figure 11 (Part 9 of 42). MQCHEXIT.C

```
          {
            *exitBufAddr = buf;
            *exitBufLength = MQS_HAM_BUFLEN;

            status = InitHAM( &mqhamMsg );

            memcpy(&origMqhamMsg, &mqhamMsg,sizeof(MQ_HAM));

            status = BuildHAM( buf, &bufLen, &mqhamMsg );
            if ( status != 0 )
            {
              WriteLogFile( 57, 'E',
               "Error building the HAM. Error code: 0x%04lX\n", status);
              pChlExParms->ExitResponse = MQXCC_SUPPRESS_FUNCTION;
              pChlExParms->ExitResponse2 = MQXR2_USE_EXIT_BUFFER;
            }
            else
            {
              *dataLength = bufLen;
              pChlExParms->ExitResponse =
                           MQXCC_SEND_AND_REQUEST_SEC_MSG;
              pChlExParms->ExitResponse2 = MQXR2_USE_EXIT_BUFFER;
              WriteLogFile( 606, 'D',
               "return(MQXCC_SEND_AND_REQUEST_SEC_MSG)\n" );
             }
          }
          break;
      case MQCHT_SERVER:
  WriteLogFile( 623, 'D', "Channel type = MQCHT_SERVER\n" );
          break;
      case MQCHT_REQUESTER:
  WriteLogFile( 624, 'D', "Channel type = MQCHT_REQUESTER\n" );
          break;
      case MQCHT_CLNTCONN:
  WriteLogFile( 625, 'D', "Channel type = MQCHT_CLNTCONN\n" );
          break;
      default:
  WriteLogFile( 67, 'E', "Channel type = (unknown type)\n" );

  }                    /* endswitch ChannelType in MQXR_INIT_SEC */
  break;
```

Figure 11 (Part 10 of 42). MQCHEXIT.C

```
        case MQXR_SEC_MSG:
            WriteLogFile( 630, 'D', "ExitReason = MQXR_SEC_MSG\n" );
            switch ( pChDef->ChannelType )
            {
                  case MQCHT_SERVER:
               WriteLogFile( 623, 'D', "Channel type = MQCHT_SERVER\n" );
                  // break;
                 case MQCHT_RECEIVER:
               WriteLogFile( 622, 'D', "Channel type = MQCHT_RECEIVER\n" );
                status = ReadAnswerHAM( &mqhamMsg, pagBuf, *dataLength );
                    if ( status != 0 )
                    {
                        WriteLogFile( 32, 'E', "Sending channel exit
failed to pass security check. Error code: 0x%04lX\n", status
);
                        WriteLogFile( 404, 'I', "Channel will be closed
by the receiver due to security violation\n" );
                        pChlExParms->ExitResponse = MQXCC_CLOSE_CHANNEL;
                        pChlExParms->ExitResponse2 = MQXR2_USE_AGENT_BUFFER;
                        WriteLogFile( 606, 'D',
"return(MQXCC_CLOSE_CHANNEL)\n" );
                    }
                    else
                    {
                        WriteLogFile( 420, 'I', "Sending channel
exit passed security check successfully\n" );
                        pChlExParms->ExitResponse = MQXCC_OK;
                        pChlExParms->ExitResponse2 = 0L;
                        WriteLogFile( 606, 'D', "return(MQXCC_OK)\n" );
                    };
                    break;
                case MQCHT_CLNTCONN:
                    WriteLogFile( 625, 'D',
"Channel type = MQCHT_CLNTCONN\n" );
                case MQCHT_SENDER:
                    WriteLogFile( 621, 'D',
"Channel type = MQCHT_SENDER\n" );
                    status = ReceiveHAM( &mqhamMsg,  pagBuf, *dataLength );
                    if ( status != 0 )
                    {
                        WriteLogFile( 33, 'E', "Receiving channel exit
failed to pass security check. Error code: 0x%04lX\n", status
);
```

Figure 11 (Part 11 of 42). MQCHEXIT.C

```
                        WriteLogFile( 404, 'I', "Channel will be closed by
the sender due to security violation\n" );
                        pChlExParms->ExitResponse = MQXCC_CLOSE_CHANNEL;
                        pChlExParms->ExitResponse2 = MQXR2_USE_AGENT_BUFFER;
               WriteLogFile( 606, 'D', "return(MQXCC_CLOSE_CHANNEL)\n" );
                    }
                    else
                    {
                        WriteLogFile( 421, 'I', "Receiving channel exit
passed security check successfully\n" );
                        buf = AllocExitBuffer( MQS_HAM_BUFLEN );
                        if ( buf == NULL )
                        {
                            WriteLogFile( 56, 'E', "Buffer allocation
failed for ExitBuffer\n" );
                        pChlExParms->ExitResponse = MQXCC_SUPPRESS_FUNCTION;
                        pChlExParms->ExitResponse2 = MQXR2_USE_AGENT_BUFFER;
           WriteLogFile( 606, 'D', "return(MQXCC_SUPPRESS_FUNCTION)\n" );
                        }
                        else
                        {
                            *exitBufAddr = buf;
                            *exitBufLength = MQS_HAM_BUFLEN;
                            status = BuildHAM( buf, &bufLen, &mqhamMsg);
                            if ( status != 0 )
                            {
           WriteLogFile( 57, 'E', "Error building the HAM\n" );
                     pChlExParms->ExitResponse = MQXCC_SUPPRESS_FUNCTION;
                       pChlExParms->ExitResponse2 = MQXR2_USE_EXIT_BUFFER;
                                WriteLogFile( 606, 'D',
"return(MQXCC_SUPPRESS_FUNCTION)\n" );
                            }
                            else
                            {
                                *dataLength = bufLen;
                                pChlExParms->ExitResponse = MQXCC_SEND_SEC_MSG;
                                pChlExParms->ExitResponse2 = MQXR2_USE_EXIT_BUFFER;
                                WriteLogFile( 606, 'D',
"return(MQXCC_SEND_SEC_MSG)\n" );
                            }
                        }
                    }
                    break;
```

Figure 11 (Part 12 of 42). MQCHEXIT.C

```
                case MQCHT_REQUESTER:
                    WriteLogFile( 624, 'D', "Channel type = MQCHT_REQUESTER\n" );
                    break;

                case MQCHT_SVRCONN:
                    WriteLogFile( 626, 'D', "Channel type = MQCHT_SVRCONN\n" );
                    break;
                default:
                WriteLogFile( 67, 'E', "Channel type = (unknown type)\n" );
            }                   /* endswitch ChannelType */
            break;

        case MQXR_RETRY:
            WriteLogFile( 631, 'D', "ExitReason = MQXR_RETRY\n" );
            break;
        default:
            WriteLogFile( 39, 'E', "Unknown ExitReason in channel
security exit call\n");
    }
    return;
}
```

Figure 11 (Part 13 of 42). MQCHEXIT.C

```
/*
 * This is the function for the send message exit.
 */
void MQENTRY    SendMessageExit( PMQVOID channelExitParms,
                                             PMQVOID channelDef,
                                             PMQLONG dataLength,
                                             PMQLONG agBufLength,
                                             PMQVOID agBuf,
                                             PMQLONG exitBufLength,
                                             PMQPTR exitBufAddr )

{

    PMQCXP          pChlExParms = ( PMQCXP ) channelExitParms;
    PMQCD           pChDef = ( PMQCD ) channelDef;
    unsigned char   *buf;
    unsigned int    bufLen;
    int             status;


    switch ( pChlExParms->ExitReason )
    {
        case MQXR_INIT:
            WriteLogFile( 630, 'D', "ExitReason = MQXR_INIT\n" );
            break;
        case MQXR_TERM:
            WriteLogFile( 631, 'D', "ExitReason = MQXR_TERM\n" );
            pChlExParms->ExitResponse = MQXCC_OK;
            pChlExParms->ExitResponse2 = MQXCC_OK;
            pChlExParms->Feedback = 0L;
            WriteLogFile( 645, 'D', "return(MQXCC_OK)\n" );
            break;
        case MQXR_MSG:
            WriteLogFile( 640, 'D', "ExitReason = MQXR_MSG\n" );
            break;
        case MQXR_XMIT:
            WriteLogFile( 650, 'D', "ExitReason = MQXR_XMIT\n" );
            switch ( pChDef->ChannelType )
            {
                case MQCHT_CLNTCONN:
               WriteLogFile( 625, 'D', "Channel type = MQCHT_CLNTCONN\n" );
```

Figure 11 (Part 14 of 42). MQCHEXIT.C

```
                case MQCHT_SENDER:
                    WriteLogFile( 621, 'D', "Channel type = MQCHT_SENDER\n" );
                    if ( *dataLength < 300 )
                    {
                        pChlExParms->ExitResponse = MQXCC_OK;
                        pChlExParms->ExitResponse2 = MQXR2_USE_AGENT_BUFFER;
                        WriteLogFile( 666, 'D', "ExitResponse2 =
MQXR2_USE_AGENT_BUFFER\n" );
                        WriteLogFile( 606, 'D', "return(MQXCC_OK)\n" );
                        break;
                    }
                    buf = BuildBufferToSendMSG( &bufLen, &status,
agBuf, *dataLength);
                    if ( status == 0 )
                    {
                        if ( buf )
                        {
                            *exitBufAddr = buf;
                            *exitBufLength = bufLen;

                            *dataLength = bufLen;
                            WriteLogFile( 632, 'D', "Sending buffer
length: %d\n", *dataLength);

                    pChlExParms->ExitResponse = MQXCC_OK;
                    pChlExParms->ExitResponse2 = MQXR2_USE_EXIT_BUFFER;
                            WriteLogFile( 666, 'D', "ExitResponse2 =
MQXR2_USE_EXIT_BUFFER\n" );
                            WriteLogFile( 606, 'D', "return(MQXCC_OK)\n" );
                        }
                        else
                        {
                            pChlExParms->ExitResponse = MQXCC_OK;
                            pChlExParms->ExitResponse2 =
MQXR2_USE_AGENT_BUFFER;
                            WriteLogFile( 666, 'D', "ExitResponse2 =
MQXR2_USE_AGENT_BUFFER\n" );
                            WriteLogFile( 606, 'D', "return(MQXCC_OK)\n" );
                        }
                    }
                    break;
```

Figure 11 (Part 15 of 42). MQCHEXIT.C

```
                case MQCHT_SERVER:
            WriteLogFile( 623, 'D', "Channel type = MQCHT_SERVER\n" );
                    break;
                case MQCHT_RECEIVER:
            WriteLogFile( 622, 'D', "Channel type = MQCHT_RECEIVER\n" );
                    break;
                case MQCHT_REQUESTER:
               WriteLogFile( 624, 'D', "Channel type = MQCHT_REQUESTER\n" );
                    break;

                case MQCHT_SVRCONN:
               WriteLogFile( 626, 'D', "Channel type = MQCHT_SVRCONN\n" );
                    break;
                default:
               WriteLogFile( 67, 'E', "Channel type = (unknown type)\n" );
            }                   /* endswitch ChannelType */
            break;
        case MQXR_SEC_MSG:
            WriteLogFile( 651, 'D', "ExitReason = MQXR_SEC_MSG\n" );
            break;
        case MQXR_INIT_SEC:
            WriteLogFile( 652, 'D', "ExitReason = MQXR_INIT_SEC\n" );
            break;
        case MQXR_RETRY:
            WriteLogFile( 653, 'D', "ExitReason = MQXR_RETRY\n" );
            break;
        default:
            WriteLogFile( 40, 'E', "Unknown ExitReason in send
message exit call\n");
    }
}
```

Figure 11 (Part 16 of 42). MQCHEXIT.C

```
/*
 * This is the function for the receive message exit
 */
void MQENTRY    ReceiveMessageExit( PMQVOID channelExitParms,
                                    PMQVOID channelDef,
                                    PMQLONG dataLength,
                                    PMQLONG agBufLength,
                                    PMQVOID agBuf,
                                    PMQLONG exitBufLength,
                                    PMQPTR exitBufAddr )

{

    PMQCXP          pChlExParms     = ( PMQCXP ) channelExitParms;
    PMQCD           pChDef          = ( PMQCD ) channelDef;
    unsigned char   *buf;
    unsigned int    bufLen;
    int             status;

    switch ( pChlExParms->ExitReason )
    {
        case MQXR_INIT:
            WriteLogFile( 660, 'D', "ExitReason = MQXR_INIT\n" );
            break;
        case MQXR_TERM:
            WriteLogFile( 661, 'D', "ExitReason = MQXR_TERM\n" );
            pChlExParms->ExitResponse = MQXCC_OK;
            pChlExParms->ExitResponse2 = MQXCC_OK;
            pChlExParms->Feedback = 0L;
            WriteLogFile( 650, 'D', "return(MQXCC_OK)\n" );
            break;
        case MQXR_MSG:
            WriteLogFile( 662, 'D', "ExitReason = MQXR_MSG\n" );
            break;
```

Figure 11 (Part 17 of 42). MQCHEXIT.C

```
        case MQXR_XMIT:
            WriteLogFile( 663, 'D', "ExitReason = MQXR_XMIT\n" );
            switch ( pChDef->ChannelType )
            {
                case MQCHT_SENDER:
               WriteLogFile( 621, 'D', "Channel type = MQCHT_SENDER\n" );
                    break;
                case MQCHT_SERVER:
               WriteLogFile( 623, 'D', "Channel type = MQCHT_SERVER\n" );
                    break;
                 case MQCHT_SVRCONN:
               WriteLogFile( 626, 'D', "Channel type = MQCHT_SVRCONN\n" );
//                    break;
                case MQCHT_RECEIVER:

               WriteLogFile( 622, 'D', "Channel type = MQCHT_RECEIVER\n" );
               WriteLogFile( 632, 'D', "Receive buffer length: %d\n",
*dataLength);

                    if ( *dataLength < 300 )
                    {
                        pChlExParms->ExitResponse = MQXCC_OK;
                        pChlExParms->ExitResponse2 = MQXR2_USE_AGENT_BUFFER;
                        WriteLogFile( 666, 'D', "ExitResponse2 =
MQXR2_USE_AGENT_BUFFER\n" );
                        WriteLogFile( 606, 'D', "return(MQXCC_OK)\n" );
                        break;
                    }
              buf = ReceiveMSGBuffer( &bufLen, agBuf,&status, *dataLength);
                    if ( status == 0 )
                    {
                        if ( buf )
                        {
                            *exitBufAddr = buf;
                            *exitBufLength = bufLen;
                            *dataLength = bufLen;
                      WriteLogFile( 632, 'D', "Sending buffer length: %d\n",
*dataLength);
                            pChlExParms->ExitResponse = MQXCC_OK;
                            pChlExParms->ExitResponse2 = MQXR2_USE_EXIT_BUFFER;
                            WriteLogFile( 666, 'D', "ExitResponse2 =
MQXR2_USE_EXIT_BUFFER\n" );
                            WriteLogFile( 606, 'D', "return(MQXCC_OK)\n" );
                        }
```

Figure 11 (Part 18 of 42). MQCHEXIT.C

```
                        else
                        {
                            pChlExParms->ExitResponse = MQXCC_OK;
                            pChlExParms->ExitResponse2 =
MQXR2_USE_AGENT_BUFFER;
                            WriteLogFile( 666, 'D', "ExitResponse2 =
MQXR2_USE_AGENT_BUFFER\n" );
                            WriteLogFile( 606, 'D', "return(MQXCC_OK)\n" );
                        }
                    }
                    break;
                case MQCHT_REQUESTER:
             WriteLogFile( 624, 'D', "Channel type = MQCHT_REQUESTER\n" );
                    break;
                case MQCHT_CLNTCONN:
             WriteLogFile( 625, 'D', "Channel type = MQCHT_CLNTCONN\n" );
                    break;
                default:
       WriteLogFile( 67, 'E', "Channel type = (unknown type       )\n" );
            }                   /* endswitch ChannelType */
            break;
        case MQXR_SEC_MSG:
            WriteLogFile( 664, 'D', "ExitReason = MQXR_SEC_MSG\n" );
            break;
        case MQXR_INIT_SEC:
            WriteLogFile( 665, 'D', "ExitReason = MQXR_INIT_SEC\n" );
            break;
        case MQXR_RETRY:
            WriteLogFile( 666, 'D', "ExitReason = MQXR_RETRY\n" );
            break;
        default:
            WriteLogFile( 41, 'E', "Unknown ExitReason in receive
message exit call\n");
    }                           /* endswitch ExitReason for ExitType
                                 * MQXT_CHANNEL_MSG_EXIT */
}
```

Figure 11 (Part 19 of 42). MQCHEXIT.C

```
void            GetInfo( PMQCD pChDef)
{

    MakeCString( channeldata.channel, pChDef->ChannelName,
MQ_CHANNEL_NAME_LENGTH );

    /* In case of a client connection channel we have no queue manager name
     * We take the name provided in the MsgUserData field */
    if ( pChDef->QMgrName[0] != 0x00 && pChDef->QMgrName[0] != ' ' )
   MakeCString( qmgrdata.qmgr, pChDef->QMgrName,  MQ_Q_MGR_NAME_LENGTH );
    else
        MakeCString( qmgrdata.qmgr, pChDef->SendUserData, 32);


    /* Get hostname*/
    if ( GetHostName(hostName, &hostNameLen) != 0 )
        WriteLogFile(56,'E',"Function GetHostName() was unsuccessful\n");

    if ( memcmp(pChDef->SecurityUserData,"DEBUG",5) == 0 )
        doDebugging = 1;
    else
        doDebugging = 0;

    if ( GetEnvironmentVariable(LOGFILE_PATH_VAR, logFilePath,256) == 0 )
    {
       WriteLogFile( 43, 'E',"Environment variable %s not defined\n",
LOGFILE_PATH_VAR);
    }

    if ( GetEnvironmentVariable(KEYFILE_PATH_VAR, keyFilePath,256) == 0 )
    {
       WriteLogFile( 44, 'E',"Environment variable %s not defined\n",
KEYFILE_PATH_VAR);
    }
}
```

Figure 11 (Part 20 of 42). MQCHEXIT.C

```
void WriteLogFile( int msgid, unsigned char msgtype, unsigned char *msgtxt, ... )
{
    time_t              ltime;
    struct tm           *newtime;
    static int          firstTime = 1;
    va_list   args;

    time(&ltime);
    newtime = localtime(&ltime);

    if ( firstTime )
    {
        firstTime = 0;
  WriteLogFile( 400, 'I', "*********** ChannelExit started ***********\n");
    printf("MQS9400I %s WTO5219A MQCHEXIT ChannelExit started\n",hostName);
    }

    if ( doDebugging == 0 && msgtype == 'D' ) return;

    if ( logFilePtr != stdout )
        if ( ( logFilePtr = fopen(logFileName,"a+") ) == NULL )
        {
            logFilePtr = stdout;
            WriteLogFile( 40, 'E',"Error opening logfile %s.\n",logFileName);
            WriteLogFile( 441, 'I',"Using stdout for log messages\n");
            return;
        }
    fprintf( logFilePtr, "%02d%03d %02d:%02d:%02d ",newtime->tm_year,
       newtime->tm_yday, newtime->tm_hour, newtime->tm_min, newtime->tm_sec);
    fprintf( logFilePtr, "MQS9%03d%c %s %s MQCHEXIT ",
             msgid, msgtype, hostName, qmgrdata.qmgr);
    va_start( args, msgtxt );
    vfprintf( logFilePtr, msgtxt, args);
    va_end (args);
    if ( logFilePtr != stdout ) fclose(logFilePtr);
    else fflush(stdout);

    #ifdef _MVS
            wto( errstr );
    #endif
}
```

Figure 11 (Part 21 of 42). MQCHEXIT.C

```
/* Builds the HAM (Handshaking Authentification message) to transmit to
 * the sending queue manager's channelexit.
 * Return the len of the HAM.
 */
int             BuildHAM( char *buf , unsigned int *bufLen, MQ_HAM *mqham)
{
    int         status;

    WriteLogFile( 700, 'D', "Encrypting the Handshaking Authentication Message\n" );
    status = EncryptHAM( mqham, &partnerpubk, &qmgrdata.privk );
    if ( status != 0 )
        return status;

    WriteLogFile( 705, 'D', "Building the buffer to send the HAM\n" );
    status = BuildBufferToSendHAM( buf, bufLen, mqham );

    return status;
}


/* Sending queue manager's channel security exit has to read HAM and to
answer properly.
 * If ok it returns 0, otherwise (e.g. if HAM is not in proper format)
it returns -1.
 */
int             ReceiveHAM( MQ_HAM *mqham, char * buf, unsigned int buflen)
{
    int     status;

    if ( buf == NULL )
        return -1;

    WriteLogFile( 720, 'D', "Receiving the Handshaking Authentication
Message\n" );
    status = ReadHAM( mqham, buf, buflen );
    if ( status != 0 )
        return status;

    WriteLogFile( 725, 'D', "Verifying the HAM\n" );
    status = VerifyHAM( mqham, &partnerpubk, &qmgrdata.privk );

    return status;
}
```

Figure 11 (Part 22 of 42). MQCHEXIT.C

```
int          ReadAnswerHAM( MQ_HAM *mqham, char *buf, unsigned int buflen)
{
    int         status;


    if ( buf == NULL )
        return -1;


    WriteLogFile( 730, 'D',
     "Receiving the answer to the Handshaking Authentication Message\n" );
    status = ReadHAM( mqham, buf, buflen );
    if ( status != 0 )
        return status;

    WriteLogFile( 735, 'D', "Verifying the answer to the HAM\n" );
    status = VerifyAnswerHAM( mqham, &origMqhamMsg, &partnerpubk, &qmgrdata.privk );
    if ( status != 0 )
        return status;

 if ( memcmp( mqham->hambuf, origMqhamMsg.hambuf,  mqham->hambuflen) == 0 )
        return 0;
    else
        return -1;

    return 0;
}


unsigned char           *AllocExitBuffer( int len )
{

    unsigned char           *buf;

    buf = ( unsigned char * ) malloc( len );

    if ( buf == NULL )
        return NULL;

    return buf;
}
```

Figure 11 (Part 23 of 42). MQCHEXIT.C

```
void            FreeExitBuffer(PMQPTR exitBufAddr, PMQLONG exitBufLength )
{

    if ( exitBufAddr != NULL && *exitBufAddr != NULL )
    {
        free( *exitBufAddr );
        *exitBufAddr = NULL;
        *exitBufLength = 0L;
    }
}

int ReadKeysFromFiles(void)
{
    int status;


    if ( ( status = ReadPrivateKey() ) != 0 )
        return status;

    WriteLogFile( 700, 'D', "Reading private keys was successful\n" );

    if ( ( status = ReadPublicKey() ) != 0 )
        return status;
    WriteLogFile( 701, 'D', "Reading public keys was successful\n" );

    return 0;
}
int ReadPrivateKey(void)
{
    FILE            *fptr;
    unsigned char   qmname[MQ_Q_MGR_NAME_LENGTH];




    /* Build name of file which does contain the private key */
    strcpy(privKeyFileName,keyFilePath);
    strcat(privKeyFileName,qmgrdata.qmgr);
    strcat(privKeyFileName,".PRV");

    WriteLogFile( 690, 'D', "Opening private key file\n" );
    fptr = fopen(privKeyFileName,"rb");
    if ( fptr == NULL )
        return 2;
```

Figure 11 (Part 24 of 42). MQCHEXIT.C

```
    /* Read queue manager's name from file */
    fread( tmpBuf, QMGR_NAME_EYE_CATCHER_SIZE, 1, fptr);
    if (memcmp(tmpBuf,QMGR_NAME_EYE_CATCHER,QMGR_NAME_EYE_CATCHER_SIZE) != 0)
        return 4;

    fread( qmname, MQ_Q_MGR_NAME_LENGTH, 1, fptr );

    /* Check if name is ok */
    if ( memcmp(qmgrdata.qmgr, qmname, strlen(qmgrdata.qmgr) ) != 0 )
        return 6;


 /* Read the CSSD's public key from file */
    fread( tmpBuf, CSSD_PUBK_EYE_CATCHER_SIZE, 1, fptr);
    if (memcmp(tmpBuf,CSSD_PUBK_EYE_CATCHER,CSSD_PUBK_EYE_CATCHER_SIZE) != 0)
        return 8;
 fread( & cssdPubk, sizeof(R_RSA_PUBLIC_KEY), 1, fptr );


 /* Read the queue manager's public key from file */
    fread( tmpBuf, QMGR_PUBK_EYE_CATCHER_SIZE, 1, fptr);
    if (memcmp(tmpBuf,QMGR_PUBK_EYE_CATCHER,QMGR_PUBK_EYE_CATCHER_SIZE) != 0)
        return 10;
 fread( &qmgrdata.pubk, sizeof(R_RSA_PUBLIC_KEY), 1, fptr );


 /* Read the queue manager's private key from file */
    fread( tmpBuf, QMGR_PRIVK_EYE_CATCHER_SIZE, 1, fptr);
    if (memcmp(tmpBuf,QMGR_PRIVK_EYE_CATCHER,QMGR_PRIVK_EYE_CATCHER_SIZE) != 0)
            return 12;
 fread( &qmgrdata.privk, sizeof(R_RSA_PRIVATE_KEY), 1, fptr );

    fclose(fptr);


    return 0;
}
```

Figure 11 (Part 25 of 42). MQCHEXIT.C

```
int ReadPublicKey(void)
{
    FILE    *fptr;
    unsigned char           qmname[MQ_Q_MGR_NAME_LENGTH];
    MQCHAR                  channelname[MQ_CHANNEL_NAME_LENGTH];
    int                     found = 0;
    unsigned char   signature[MAX_SIGNATURE_LEN];
 unsigned int   signatureLen;
    R_SIGNATURE_CTX         context;
    int                     status;
    int                     offset;


    /* Build name of file which does contain the private key */
    strcpy(pubKeyFileName,keyFilePath);
    strcat(pubKeyFileName,qmgrdata.qmgr);
    strcat(pubKeyFileName,".PUB");

    WriteLogFile( 691, 'D', "Opening public key file\n" );

    fptr = fopen(pubKeyFileName,"rb");
    if ( fptr == NULL )
        return 32;


    while ( fread(tmpBuf, CHANNEL_NAME_EYE_CATCHER_SIZE, 1, fptr ) )
    {
        if (  memcmp( tmpBuf, CHANNEL_NAME_EYE_CATCHER ,
CHANNEL_NAME_EYE_CATCHER_SIZE ) != 0 )
            return 34;
        fread(channelname, MQ_CHANNEL_NAME_LENGTH, 1, fptr );
        if ( memcmp(channelname, channeldata.channel,
strlen(channeldata.channel) ) == 0 )
        {
            /* We found the channel in the file
             * Reading partner info: name of qmgr and public key now. */

            fread( tmpBuf, QMGR_NAME_EYE_CATCHER_SIZE, 1, fptr);
            if (  memcmp( tmpBuf, QMGR_NAME_EYE_CATCHER ,
QMGR_NAME_EYE_CATCHER_SIZE ) != 0 )
                return 36;
            fread( partnername, MQ_Q_MGR_NAME_LENGTH, 1, fptr);
```

Figure 11 (Part 26 of 42). MQCHEXIT.C

```
            fread( tmpBuf, QMGR_PUBK_EYE_CATCHER_SIZE, 1, fptr);
            if (  memcmp( tmpBuf, QMGR_PUBK_EYE_CATCHER ,
QMGR_PUBK_EYE_CATCHER_SIZE ) != 0 )
                return 38;
            fread(&partnerpubk, sizeof(R_RSA_PUBLIC_KEY), 1, fptr);

      /* Reading the digital signature to verify the info just gotten */
            fread( tmpBuf, DIGSIG_EYE_CATCHER_SIZE, 1, fptr);
            if (  memcmp( tmpBuf, DIGSIG_EYE_CATCHER ,
DIGSIG_EYE_CATCHER_SIZE ) != 0 )
                return 40;
            fread( &signatureLen, sizeof(unsigned int), 1, fptr);
            fread( signature, MAX_SIGNATURE_LEN, 1, fptr);

            /* We will verify the data now  */
            R_VerifyInit(&context, DA_MD5);
            R_VerifyUpdate(&context, channelname, MQ_CHANNEL_NAME_LENGTH);
            R_VerifyUpdate(&context, partnername, MQ_Q_MGR_NAME_LENGTH);
            R_VerifyUpdate(&context, (unsigned char *) &partnerpubk,
sizeof(R_RSA_PUBLIC_KEY));
            status = R_VerifyFinal(&context, signature, signatureLen, &cssdPubk);
            if ( status != 0 )
                return status;

            found = 1; /* Indicate success */
            break;
        }
        else
        {
            /* point to next record */
            offset = QMGR_NAME_EYE_CATCHER_SIZE + MQ_Q_MGR_NAME_LENGTH +
                        QMGR_PUBK_EYE_CATCHER_SIZE + sizeof(R_RSA_PUBLIC_KEY) +
                        DIGSIG_EYE_CATCHER_SIZE + sizeof(unsigned int) +
                        MAX_SIGNATURE_LEN;
            fseek(fptr, offset, SEEK_CUR );
        }
    }
    if ( found == 0 ) return 50;
    fclose(fptr);
    return 0;
}
```

Figure 11 (Part 27 of 42). MQCHEXIT.C

```
void            MQStart( void )
{
}



/* Copy from a blank-filled array to a zero-terminated string */
void            MakeCString( char *zt, char *bf, int len )
{
    int             i;


    for ( i = 0; bf[i] != ' ' && i < len; i++ )
        zt[i] = bf[i];
    zt[i] = 0;
}



void            BuildLogFileName(void)
{

    /* Build logfile name */
    strcpy(logFileName, logFilePath);
    strcat(logFileName, qmgrdata.qmgr);
    strcat(logFileName, ".MQCHEXIT.LOG");

    return;
}


int             InitHAM( MQ_HAM * mqham )
{
    FillRandomBuffer16( mqham->hambuf );
    mqham->hambuflen = 16;

    return 0;
}
```

Figure 11 (Part 28 of 42). MQCHEXIT.C

```
/* Caller has to provide a 32 bytes outbuf for the HAM to be put to.
 */
int             EncryptHAM( MQ_HAM               *mqham,
                            R_RSA_PUBLIC_KEY     *pubk,
                            R_RSA_PRIVATE_KEY    *privk )
{
    int status;
    status = EncryptBuffer( mqham->hamEncryptedBuf,
                            &( mqham->hamEncryptedBufLen ),
                            mqham->hambuf,
                            mqham->hambuflen,
                            MQ_DO_NO_COMPRESS,
                            pubk,
                            privk,
                            &( mqham->mqdigsig ) );
    return status;
}

int             BuildBufferToSendHAM( unsigned char *outbuf,
                                      unsigned int *outbuflen,
                                      MQ_HAM * mqham )
{
    memcpy( outbuf, HAM_EYE_CATCHER, HAM_EYE_CATCHER_SIZE);
    *outbuflen = HAM_EYE_CATCHER_SIZE;

    memcpy( outbuf + *outbuflen, &( mqham->mqdigsig ), sizeof
( MQ_DIGITAL_SIGNATURE ) );
    *outbuflen += sizeof( MQ_DIGITAL_SIGNATURE );

    memcpy( outbuf + *outbuflen, &( mqham->hamEncryptedBufLen ),
sizeof( unsigned int ) );
    *outbuflen += sizeof( unsigned int );

    memcpy( outbuf + *outbuflen, mqham->hamEncryptedBuf,
mqham->hamEncryptedBufLen );
    *outbuflen += mqham->hamEncryptedBufLen;

    WriteLogFile( 607, 'D', "Sending %d bytes. Encypted buffer length %d\n",
                *outbuflen, mqham->hamEncryptedBufLen );
    return 0;
}
```

Figure 11 (Part 29 of 42). MQCHEXIT.C

```
/* Returns an encrypted digital signature and the encrypted buffer in outbuf.
 * Caller must provide a buffer large enough to hold the data.
 * The function itself returns the number of bytes written to outbuf in oubbuflen.
 * ATTENTION: Currently compression is not implemented.
 */
int              EncryptBuffer( unsigned char *outbuf,
                                                  unsigned int *outbuflen,
                                                  unsigned char *inbuf,
                                                  unsigned int inbuflen,
                                                  unsigned int compressflag,
                                                  R_RSA_PUBLIC_KEY * recpubk,
                                               R_RSA_PRIVATE_KEY * sdrprivk,
                                           MQ_DIGITAL_SIGNATURE * mqdigsig )
{
    R_RANDOM_STRUCT random;
    R_ENVELOPE_CTX  context;
    int             status;
    unsigned char   encryptedKey[MAX_ENCRYPTED_KEY_LEN];
    unsigned char  *encryptedKeys[1];
    unsigned int    encryptedKeyLen;
    unsigned char   iv[8];
    unsigned char   signature[MAX_SIGNATURE_LEN];
    unsigned int    signatureLen;
    unsigned int    outlen;
    unsigned char   *partOut;
    unsigned int    partOutLen;
    unsigned char   *partIn;
    unsigned int    partInLen;

    status = 0;

    R_RandomCreate( &random );
    encryptedKeys[0] = encryptedKey;
    R_memset( (unsigned char *) &context, 0, sizeof(R_ENVELOPE_CTX));
    /* Digital signature of block */
    status = R_SignBlock( signature, &signatureLen, inbuf, inbuflen, DA_MD5,
sdrprivk );

    mqdigsig->signatureLen = signatureLen;
    memcpy( mqdigsig->signature, signature, signatureLen );
```

Figure 11 (Part 30 of 42). MQCHEXIT.C

```
    status = R_SealInit( &context,
                         encryptedKeys,
                         &encryptedKeyLen,
                         iv,
                         1,     /* number of public keys */
                         &recpubk,
                         EA_DES_EDE3_CBC,
                         &random );
    if ( status != 0 )
        return status;

    partIn          = inbuf;
    partInLen       = 2048;

    partOut         = outbuf;
    partOutLen      = 0;

    outlen      = 0;

    while( inbuflen > 0 )
    {
  if ( partInLen > inbuflen )
   partInLen = inbuflen;

        status = R_SealUpdate( &context, partOut, &partOutLen, partIn, partInLen);
        if ( status != 0 )
            return status;

        outlen += partOutLen;
        partOut += partOutLen;

        inbuflen -= partInLen;
        partIn  += partInLen;

    }
```

Figure 11 (Part 31 of 42). MQCHEXIT.C

```
    status = R_SealFinal( &context, partOut, &partOutLen );
    if ( status != 0 )
        return status;
    outlen  += partOutLen;
    partOut += partOutLen;

    *outbuflen  = outlen;



    R_RandomFinal( &random );

    mqdigsig->encryptedKeyLen = encryptedKeyLen;
    memcpy( mqdigsig->encryptedKey, encryptedKey, encryptedKeyLen );
    memcpy( mqdigsig->iv, iv, sizeof( iv ) );



    /* Clear sensitive information. */
    R_memset( ( POINTER ) & context, 0, sizeof( context ) );
    R_memset( ( POINTER ) inbuf, 0, inbuflen );

    return 0;
}
```

Figure 11 (Part 32 of 42). MQCHEXIT.C

```
/* Checks if the encrypted buffer inbuf goes along with the digital signature
 * mqdigsig and decrypts the content of inbuf to outbuf. The number of bytes
 * written to outbuf are returned in outbuflen.
 * Caller must provide a buffer large enough to hold thee output data.
 */
int             DecryptAndVerifyBuffer( unsigned char *outbuf,
                                         unsigned int *outbuflen,
                                         unsigned char *inbuf,
                                         unsigned int inbuflen,
                                         R_RSA_PUBLIC_KEY * sdrpubk,
                                       R_RSA_PRIVATE_KEY * recprivk,
                                 MQ_DIGITAL_SIGNATURE * mqdigsig )
{
    R_RANDOM_STRUCT random;
    R_ENVELOPE_CTX  context;
    int             status;
    unsigned char   encryptedKey[MAX_ENCRYPTED_KEY_LEN];
    unsigned char  *encryptedKeys[1];
    unsigned int    encryptedKeyLen;
    unsigned char   iv[8];
    unsigned int    inlen;
    unsigned int    outlen;
    ULONG           magic;
    unsigned char   *partOut;
    unsigned int    partOutLen;
    unsigned char   *partIn;
    unsigned int    partInLen;
    USHORT          slen;

    status = 0;
    status = R_OpenInit( &context,
                         EA_DES_EDE3_CBC,
                         mqdigsig->encryptedKey,
                         mqdigsig->encryptedKeyLen,
                         mqdigsig->iv,
                         recprivk );
    if ( status != 0 ) return status;

    partIn          = inbuf;
    partInLen       = 1024;
    partOut         = outbuf;
    partOutLen      = 0;
    outlen          = 0;
```

Figure 11 (Part 33 of 42). MQCHEXIT.C

```
 while(  inbuflen > 0 )
    {
        if ( partInLen > inbuflen )
   partInLen = inbuflen;

        status = R_OpenUpdate( &context, partOut, &partOutLen, partIn,
partInLen );
     if ( status != 0 )
            return status;

        outlen += partOutLen;
     partOut += partOutLen;

     inbuflen -= partInLen;
        partIn  += partInLen;

    };

 status = R_OpenFinal( &context, partOut, &partOutLen );
    if ( status != 0 )
        return status;
    outlen  += partOutLen;
    partOut += partOutLen;

    *outbuflen  = outlen;



    /* Verify digital signature of block */
    status = R_VerifyBlockSignature( outbuf, *outbuflen,
mqdigsig->signature, mqdigsig->signatureLen, DA_MD5, sdrpubk );
    if ( status != 0 )
        return status;

    /* Clear sensitive information. */
    R_memset( ( POINTER ) & context, 0, sizeof( context ) );


    return 0;
}
```

Figure 11 (Part 34 of 42). MQCHEXIT.C

```
/* Fills  a buffer of len bytes with random values. Buffer must be
provided by caller.
 * Returns 0 if successful, -1 otherwise.
 */
int             FillRandomBuffer( unsigned char *outbuf, int len )
{
    R_RANDOM_STRUCT randomstruct;

    if ( outbuf == 0 )
        return -1;

    R_RandomCreate( &randomstruct );
    R_GenerateBytes( outbuf, len, &randomstruct );
    R_RandomFinal( &randomstruct );

    return 0;
}


/* Fills a 16 bytes length buffer with random values. Buffer must be
provided by caller.
 * Returns 0 if successful, -1 otherwise.
 */
int             FillRandomBuffer16( unsigned char *outbuf )
{
    return FillRandomBuffer(outbuf,16);
}
```

Figure 11 (Part 35 of 42). MQCHEXIT.C

```
int             ReadHAM( MQ_HAM * mqham,
                              unsigned char *rcvrbuf,
                              unsigned int rcvrbuflen )
{
    int             offset;


    memcpy( tmpBuf, rcvrbuf, HAM_EYE_CATCHER_SIZE );
    offset = HAM_EYE_CATCHER_SIZE;
    if ( memcmp(tmpBuf, HAM_EYE_CATCHER, HAM_EYE_CATCHER_SIZE) != 0 )
        return 100;


    memcpy( &( mqham->mqdigsig ), rcvrbuf+ offset,
sizeof( MQ_DIGITAL_SIGNATURE ) );
    offset += sizeof( MQ_DIGITAL_SIGNATURE );

    memcpy( &( mqham->hamEncryptedBufLen ), rcvrbuf + offset,
sizeof( unsigned int ) );
    offset += sizeof( unsigned int );

    memcpy( mqham->hamEncryptedBuf, rcvrbuf + offset,
mqham->hamEncryptedBufLen );


    WriteLogFile(700, 'D', "%d bytes read. Encypted buffer length is %d\n",
                        offset + mqham->hamEncryptedBufLen,
mqham->hamEncryptedBufLen);
    return 0;
}
```

Figure 11 (Part 36 of 42). MQCHEXIT.C

```
int             VerifyHAM( MQ_HAM * mqham,
                           R_RSA_PUBLIC_KEY * pubk,
                           R_RSA_PRIVATE_KEY * privk )
{
    int             status;

    status = DecryptAndVerifyBuffer( mqham->hambuf,
                           &( mqham->hambuflen ),
                           mqham->hamEncryptedBuf,
                           mqham->hamEncryptedBufLen,
                           pubk,
                           privk,
                           &( mqham->mqdigsig ) );

    return status;

}

int             VerifyAnswerHAM( MQ_HAM * mqham, MQ_HAM *origmqham,
                                           R_RSA_PUBLIC_KEY * sdrpubk,
                                       R_RSA_PRIVATE_KEY * rcvrprivk )
{
    int             status;

    status = VerifyHAM( mqham, sdrpubk, rcvrprivk );

    if ( status != 0 ) return status;

    /* Check that the message just decrypted is equal to what we did send in
     * the beginning of the handshaking!
     */
    if ( mqham->hambuflen != origmqham->hambuflen )
        return MQRE_HAM_MSGLEN;

    if ( memcmp(mqham->hambuf,origmqham->hambuf,origmqham->hambuflen) != 0 )
        return MQRE_HAM_MSG;

    return status;
}
```

Figure 11 (Part 37 of 42). MQCHEXIT.C

```
#define FILLER          ' '
#define HEX_PER_LINE    32

void     PrintBuff( unsigned char *out, unsigned int len )
{
    int            i, j;
    unsigned       uNum  = 0;
    if ( doDebugging == 0 ) return;
    if ( logFilePtr != stdout )
        if ( ( logFilePtr = fopen(logFileName,"a+") ) == NULL )
        {
            logFilePtr = stdout;
            WriteLogFile( 40, 'E',"Error opening logfile %s.\n",logFileName);
            WriteLogFile( 41, 'E',"Using stdout for log messages\n");
            return;
        }
    for (i = 0; i < len; i += HEX_PER_LINE)
    {
        fprintf(logFilePtr,"%5d %4x:", uNum, uNum);
        for (j = 0; j < HEX_PER_LINE; j++)
        {
            if (j % 4 == 0)
                fprintf(logFilePtr," ");
            if (j + i < len)
                fprintf(logFilePtr,"%02x", (int) out[j + i] );
            else fprintf(logFilePtr,"  ");
        }
        fprintf(logFilePtr,",");
        for (j = 0; j < HEX_PER_LINE; j++)
        {
            if (j + i < len)
            {
               if (isprint((int) out [j + i] ) == 0)
                    fprintf(logFilePtr,"%c", FILLER);
               else fprintf(logFilePtr,"%c",(int) out [j + i]);
            }
            else fprintf(logFilePtr," ");
        }
        fprintf(logFilePtr,"*\n");
        uNum += HEX_PER_LINE;
    }
    if ( logFilePtr != stdout ) fclose(logFilePtr);
    else fflush(stdout);
}
```

Figure 11 (Part 38 of 42). MQCHEXIT.C

```
unsigned char*          BuildBufferToSendMSG( unsigned int *outbuflen,
                                              int           *status,
                                              unsigned char *inbuf,
                                              unsigned int inbuflen )
{
    unsigned char               *buffer;
    unsigned char               *optr;
    unsigned int                offsetOut;
    unsigned int                offsetIn;
    unsigned int                encryptBufLen;
    MQ_DIGITAL_SIGNATURE        mqdigsig;
    int                         mallocSz;

    *status = 0;

    mallocSz = 8 + MSG_EYE_CATCHER_SIZE + sizeof( MQ_DIGITAL_SIGNATURE ) +
sizeof(unsigned int) + inbuflen  + 1 + 8;
    if (  ( buffer = (unsigned char*) malloc( mallocSz)  ) == 0 )
        return 0;


    WriteLogFile(923, 'D', "Malloc size %d\n", mallocSz);
    memcpy( buffer,inbuf,8);

    offsetOut   = 8;
    offsetIn    = 8;
    inbuflen - offsetIn;

    buffer[8]   = 0;  /* This is to indicate that data is encrypted and
should be handled by the receiver's side */
    offsetOut   += 1;

    memcpy(buffer + offsetOut, MSG_EYE_CATCHER, MSG_EYE_CATCHER_SIZE);

    offsetOut   += MSG_EYE_CATCHER_SIZE;

    optr = buffer + offsetOut + sizeof( MQ_DIGITAL_SIGNATURE ) +
sizeof(unsigned int);
```

Figure 11 (Part 39 of 42). MQCHEXIT.C

```
    *status = EncryptBuffer( optr,
                             &encryptBufLen,
                             inbuf + offsetIn,
                             inbuflen-8,
                             MQ_DO_NO_COMPRESS,
                             &partnerpubk,
                             &qmgrdata.privk,
                             &mqdigsig );
    memcpy(buffer + 9 +MSG_EYE_CATCHER_SIZE, &mqdigsig,
sizeof( MQ_DIGITAL_SIGNATURE ) );
    memcpy(buffer + 9  + MSG_EYE_CATCHER_SIZE +
sizeof( MQ_DIGITAL_SIGNATURE ), &encryptBufLen, sizeof(unsigned int) );

    *outbuflen = 8 + 1 + MSG_EYE_CATCHER_SIZE +
sizeof( MQ_DIGITAL_SIGNATURE ) + sizeof(unsigned int) + encryptBufLen;

    // PrintBuff(buffer,*outbuflen);
    return buffer;
}
```

Figure 11 (Part 40 of 42). MQCHEXIT.C

```
unsigned char*          ReceiveMSGBuffer( unsigned int *outbuflen,
                                          unsigned char *inbuf,
                                          int        *status,
                                          unsigned int inbuflen )
{
    unsigned char               *buffer;
    unsigned char               *optr;
    unsigned int                offsetOut;
    unsigned int                offsetIn;
    unsigned int                encryptBufLen;
    MQ_DIGITAL_SIGNATURE        mqdigsig;
    unsigned int                inlen;
    *status = 0;
    if ( inbuf[8] != 0 ) return 0;
    // PrintBuff(inbuf,inbuflen);

    if ( memcmp( inbuf+9, MSG_EYE_CATCHER, MSG_EYE_CATCHER_SIZE) != 0 )
        return 0;

    if (  ( buffer = (unsigned char*) malloc(  inbuflen  + 8) ) == 0 )
        return 0;

    memcpy(buffer, inbuf, 8);
    memcpy(&mqdigsig, inbuf+9+MSG_EYE_CATCHER_SIZE,
                    sizeof( MQ_DIGITAL_SIGNATURE ) );
    // PrintBuff((unsigned char*) &mqdigsig, sizeof(MQ_DIGITAL_SIGNATURE) );

    memcpy(&inlen, inbuf+ 8+1+ MSG_EYE_CATCHER_SIZE +
           sizeof( MQ_DIGITAL_SIGNATURE ), sizeof(unsigned int ) );

    WriteLogFile(890, 'D', "Dumping 'unsigned int inlen' value\n");
    PrintBuff((unsigned char*) &inlen, sizeof(unsigned int) );
    *status = DecryptAndVerifyBuffer( buffer + 8,
                                &encryptBufLen,
                                inbuf + 1 + 8 +MSG_EYE_CATCHER_SIZE +
sizeof( MQ_DIGITAL_SIGNATURE )+ sizeof(unsigned int ),
                                inlen,
                                &partnerpubk,
                                &qmgrdata.privk,
                                &mqdigsig );
    *outbuflen = encryptBufLen+8;
    //    PrintBuff(buffer, *outbuflen );
    return buffer;
}
```

Figure 11 (Part 41 of 42). MQCHEXIT.C

```
/* Returns 0 if successful, otherwise != 0.
 * If successful the hostname is returned in buf and its length in bufLen.
 * ATTENTION: The application has to be linked with WSOCK32.LIB.
 */
int  GetHostName(unsigned char *buf, unsigned int *bufLen)
{
    int status;
    WSADATA wsaData;
    WORD wVersionRequested;
    wVersionRequested = MAKEWORD(1, 1);


    status = WSAStartup(wVersionRequested, &wsaData);
    if ( status != 0)
        return status;

    status = gethostname(buf,255);
    if ( status != 0)
        return status;

    *bufLen = strlen(buf) + 1; /* we want to count the ending zero */

    return 0;
}
```

Figure 11 (Part 42 of 42). MQCHEXIT.C

Appendix B. Special Notices

This publication is intended to help application programmers to use MQSeries security exits on Windows NT. The information in this publication is not intended as the specification of any programming interfaces that are provided by Windows NT or MQSeries for Windows NT. See the PUBLICATIONS section of the IBM Programming Announcement for MQSeries for Windows NT Version 5.0 for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM′s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM′s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer′s ability to evaluate and integrate them into the customer′s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers

© Copyright IBM Corp. 1998

attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain 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.

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

AIX	IBM
MQ	MQSeries
MVS (block letters)	MVS (logo)
MVS/ESA	OS/390
RACF	S/390
System/390	

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

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

Appendix C. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

C.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see "How to Get ITSO Redbooks" on page 93.

- *MQSeries Version 5 Programming Examples*, SG24-5214
- *MQSeries Backup and Recovery*, SG24-5222

C.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
Lotus Redbooks Collection	SBOF-6899	SK2T-8039
Tivoli Redbooks Collection	SBOF-6898	SK2T-8044
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
RS/6000 Redbooks Collection (PDF Format)	SBOF-8700	SK2T-8043
Application Development Redbooks Collection	SBOF-7290	SK2T-8037

C.3 Other Publications

These publications are also relevant as further information sources:

- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries Application Programming Guide*, SC33-0807
- *MQSeries Systen Administration*, SC33-1873
- *MQSeries Intercommunication*, SC33-1872
- *MQSeries Command Reference*, SC33-1369
- *MQSeries Clients*, GC33-1632

© Copyright IBM Corp. 1998

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** http://www.redbooks.ibm.com/

  Search for, view, download or order hardcopy/CD-ROMs redbooks from the redbooks web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this redbooks site.

  Redpieces are redbooks in progress; not all redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders via e-mail including information from the redbook order form to:

	IBMMAIL	Internet
In United States:	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada:	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America:	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone Orders**

United States (toll free)	1-800-879-2755	
Canada (toll free)	1-800-IBM-4YOU	
Outside North America	(long distance charges apply)	
(+45) 4810-1320 - Danish	(+45) 4810-1220 - French	(+45) 4810-1270 - Norwegian
(+45) 4810-1420 - Dutch	(+45) 4810-1020 - German	(+45) 4810-1120 - Spanish
(+45) 4810-1540 - English	(+45) 4810-1620 - Italian	(+45) 4810-1170 - Swedish
(+45) 4810-1670 - Finnish		

This information was current at the time of publication, but is continually subject to change. The latest information for customers may be found at http://www.redbooks.ibm.com/ and for IBM employees at http://w3.itso.ibm.com/.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and redbooks by accessing the IBM Intranet Web site at http://w3.itso.ibm.com/ and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may also view redbook, residency and workshop announcements at http://inews.ibm.com/.

© Copyright IBM Corp. 1998

IBM Redbook Fax Order Form

Fax your redbook orders to:

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

Please send me the following:

Title	**Order Number**	**Quantity**

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

- Invoice to customer number

- Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

List of Abbreviations

API application program interface

CA certificate authority

CHAD channel auto-defintion

CSSD central security server database

DES Digital Encryption Standard

HAM handshaking authentication message

IBM International Business Machines Corporation

IDEA International Data Encryption Standard

ITSO International Technical Support Organization

MCA Message Channel Agent

MQ Message Queuing

MQI Message Queuing Interface

MQM MQ queue manager

MVS Multiple Virtual Storage

NSA National Security Agency

© Copyright IBM Corp. 1998

Index

A

abbreviations 19, 95
acronyms 95
activate security exit 28
AIX 30
application gateway 21
authentication 3

B

bibliography 91
big endian 32
block ciphers 4
block size 4

C

central security server database 17
certificate 8
certificate authority 8
CHAD 15
channel
  close 19
  concepts 11
  pairs 12
  types 11
channel auto-definition exit 15
channel definition 17, 23, 29
  clntconn/svrconn 30
  sender/receiver 29
channel exit 23
  chaining 13
channel exits 11
channel security exit 21
channel security server database 18
CHANNELS.INP 28
ciphers 4
ciphertext 1
client application 17, 21
CLNTCONN definition 30
compile 27
confidentiality 3
cryptographic algorithm 1, 3
cryptographic protocols 7
cryptographic software 18
cryptography 3
cryptography software 23
CSSD 17, 18, 23
CSSD.c 37
CSSD.EXE 28

D

decryption 1, 17
design-independent 17
digital encryption standard 4
digital signature 3, 5, 7
  example 7
diskette 33
DLL 27
download 23

E

eavesdropper 1
ElGamal 5
encryption 1, 17
endian 32
ExitsDefaultPath 28
export definition 27

F

fingerprints 6
firewall 21

G

goals 3

H

HAM 19
handshaking 17, 19
  in detail 24

© Copyright IBM Corp. 1998

handshaking authentication message 19
hash functions 6

I

icc 27
installation 28
integrity 3
international data encryption standard 4

K

key 2
  private and public 5
key management 8

L

legacy application 21
link 27
little endian 32
log 29

M

MCA 11
MD5 6
message 25
message digest 6
message exit 23
  description 14
  example 47
message retry exit 14
mover 11
MQCHEXIT 26
MQCHEXIT.C 47
MQCHEXIT.DLL 27
MQCHEXIT.EXP 27
MQCHEXIT.H 33
mqchexit.inp 27
MQCHEXIT.LOG 29
mqs.ini file 28
MQSCY_KEYFILEPATH 28
MQSCY_LOGFILEPATH 29
MQSEC.H 34
MQXCC_OK 25
MQXCC_SEND_AND_REQUEST_SEC_MSG 25
MQXCC_SEND_SEC_MSG 25
MQXCC_SUPRESS_FUNCTION 25
MQXR_INIT 25
MQXR_INIT_SEC 25
MQXR_SEC_MSG 25
MQXX_SUPRESS_FUNCTION 25

N

naming convention 12, 25
National Security Agency 4
non-repudiation 3
notation 2

O

objectives 3
on different platforms 27
organizational policies 21
organizational procedures 21
OS/390 31

P

password 21
plain text 1
port to AIX 30
port to OS/390 31
potential threats 3
public certificate 9
public key certificate 8
public key cryptography 5

Q

QMGRS.INP 28

R

Rabin 5
random sequence generator 6
Reaper Technologies 23
receive exit 14, 17, 23
receiver channel definition 29
response file 27

RSA 5
RSAEURO 23

S

secret key 3
secret key cryptography 4
secure communication 1
security concept 21
security exit 23
  activate 28
  description 13
  design 17
  example 47
  implementation 23
  installation 28
  message 25
security message 25
send exit 14, 17, 23
sender channel definition 29
sending a message 20
signed 8
software 18, 23
stream ciphers 4
SVRCONN definition 30

T

terminology 1
threats 3
transmission queue 12

U

user ID 21

W

Windows NT 27
winsock32.lib 27

ITSO Redbook Evaluation

MQSeries Security: Example of Using a Channel Security Exit,
SG24-5306-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and fax it to: USA International Access Code + 1 914 432 8264 or:**

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
__Customer **__Business Partner** **__Solution Developer** **__IBM employee**
__None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction ____________

Please answer the following questions:

Was this redbook published in time for your needs? Yes____ No____

If no, please explain:

____

____

____

____

What other redbooks would you like to see published?

____

____

____

Comments/Suggestions: **(THANK YOU FOR YOUR FEEDBACK!)**

____

____

____

____

____

© Copyright IBM Corp. 1998

MQSeries Security: Example of Using a Channel Security Exit, Encryption and Decryption

SG24-5306-00

SG24-5306-00
Printed in the U.S.A.