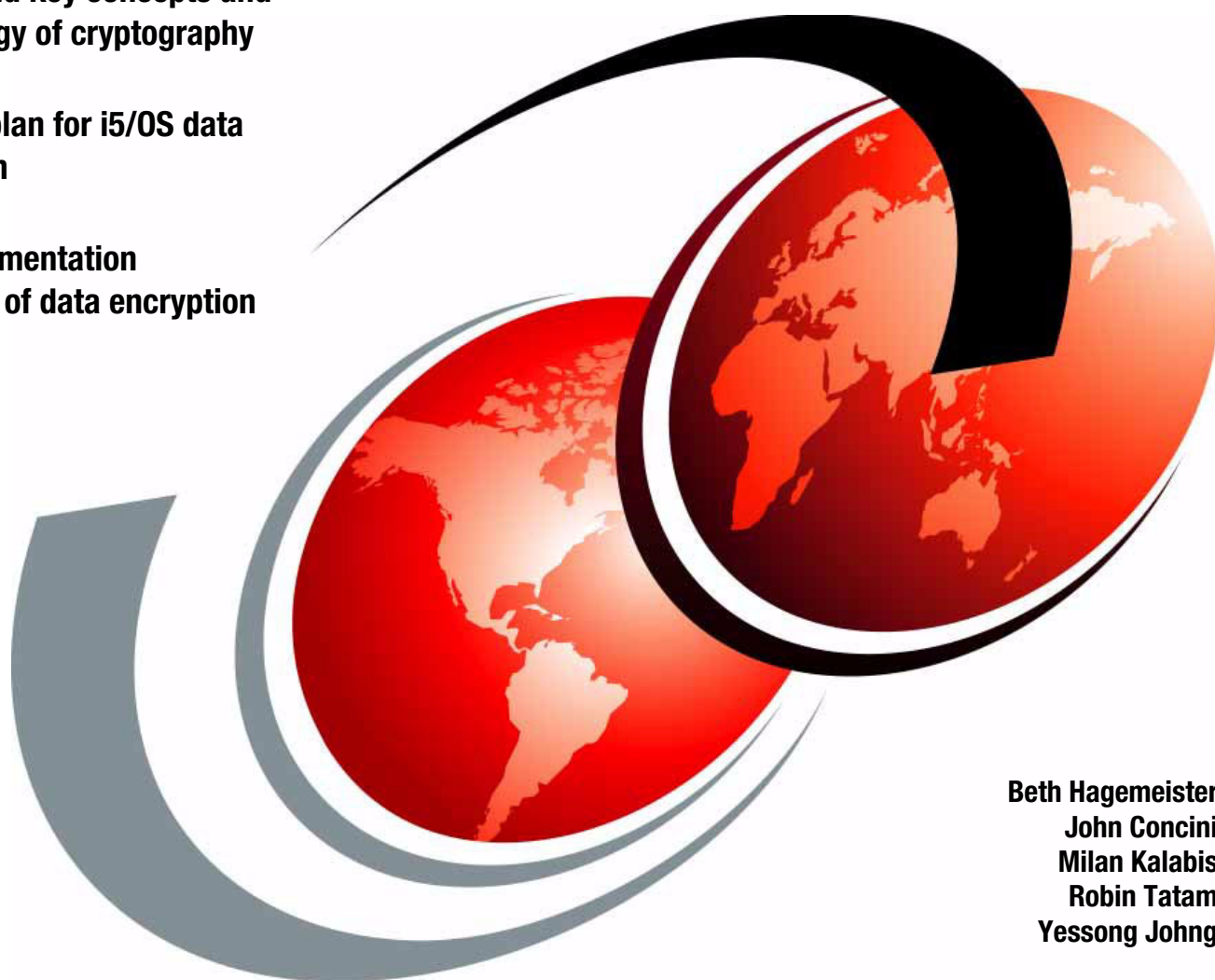


# IBM System i Security: Protecting i5/OS Data with Encryption

Understand key concepts and terminology of cryptography

Properly plan for i5/OS data encryption

See implementation scenarios of data encryption



Beth Hagemeister  
John Concini  
Milan Kalabis  
Robin Tatam  
Yessong Johng

**Red**books





International Technical Support Organization

**IBM System i Security: Protecting i5/OS Data with Encryption**

July 2008

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

**First Edition (July 2008)**

This edition applies to IBM i5/OS V5R4.

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

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

# Contents

<b>Notices</b> .....	ix
Trademarks .....	x
<b>Preface</b> .....	xi
The team that wrote this book .....	xi
Become a published author .....	xii
Comments welcome .....	xii
<b>Part 1. Introduction to data encryption</b> .....	1
<b>Chapter 1. Data encryption: the big picture</b> .....	3
1.1 Introducing encryption .....	4
1.1.1 What is data encryption .....	4
1.1.2 What drives the requirement .....	4
1.1.3 Where to protect the data .....	5
1.2 Building on top of a secure foundation .....	6
1.2.1 Only one piece of the security puzzle .....	6
1.2.2 Security is not only about privacy .....	6
1.2.3 Surely not on System i .....	7
1.2.4 Security initiatives to consider .....	7
1.2.5 A final word on traditional security .....	8
1.3 What this book is about .....	8
1.3.1 Book objectives .....	8
1.3.2 Book road map .....	9
<b>Chapter 2. Algorithms, operations, and System i implementations</b> .....	11
2.1 Cryptographic algorithms .....	12
2.1.1 Cipher algorithms .....	12
2.1.2 Key distribution algorithms .....	17
2.1.3 One-way hash algorithms .....	17
2.1.4 Random number generation algorithms .....	18
2.1.5 Summary of algorithms .....	19
2.2 Cryptographic operations .....	19
2.2.1 Data confidentiality .....	20
2.2.2 Data authentication, integrity, and non-repudiation .....	20
2.2.3 Key and random number generation .....	21
2.2.4 Financial PINs .....	21
2.2.5 Key management .....	22
2.3 System i cryptographic implementations overview .....	22
2.3.1 Cryptographic service providers .....	22
2.3.2 Cryptographic interfaces .....	23
<b>Chapter 3. Key management concepts</b> .....	25
3.1 Key management considerations .....	26
3.2 Size matters .....	26
3.3 Establishing a key value .....	28
3.3.1 Generating a key value .....	28
3.3.2 Using a known key value .....	28

3.4 Storing keys . . . . .	29
3.5 Key separation . . . . .	30
3.5.1 Key hierarchy . . . . .	30
3.5.2 Key use . . . . .	31
3.5.3 Keystore authorization . . . . .	32
3.5.4 Key management responsibilities . . . . .	32
3.6 Backing up keys . . . . .	32
3.7 Changing keys . . . . .	33
3.8 Key distribution . . . . .	34
3.9 Key destruction . . . . .	35
<b>Part 2. Planning for data encryption . . . . .</b>	<b>37</b>
<b>Chapter 4. Analyzing needs and defining scope . . . . .</b>	<b>39</b>
4.1 Needs analysis . . . . .	40
4.2 Defining the scope . . . . .	43
4.2.1 What data to protect . . . . .	43
4.2.2 Define your requirements . . . . .	44
4.2.3 Evaluate the impact of change . . . . .	45
4.2.4 Return on investment . . . . .	46
<b>Chapter 5. Managing keys on System i . . . . .</b>	<b>47</b>
5.1 Cryptographic services . . . . .	48
5.1.1 Master keys . . . . .	48
5.1.2 Keystore files . . . . .	51
5.1.3 Changing a master key . . . . .	52
5.1.4 Master key variants . . . . .	52
5.1.5 Using keys in an application . . . . .	52
5.1.6 Key distribution . . . . .	54
5.1.7 Generating keys . . . . .	55
5.1.8 Backing up keys . . . . .	55
5.2 CCA key management . . . . .	55
5.2.1 Configuring the cryptographic coprocessor . . . . .	56
5.2.2 Master keys . . . . .	57
5.2.3 Key tokens . . . . .	58
5.2.4 Keystore files . . . . .	58
5.2.5 Retain keys . . . . .	59
5.2.6 Control vectors . . . . .	59
5.2.7 Key identifier . . . . .	60
5.2.8 Key distribution . . . . .	60
5.2.9 Changing master keys . . . . .	62
5.2.10 Generating keys . . . . .	62
5.2.11 Backing up keys . . . . .	63
5.2.12 Using multiple coprocessors . . . . .	63
5.3 Roll your own key management . . . . .	64
5.4 Establishing a secure keystore environment . . . . .	65
5.4.1 Object security 101 . . . . .	65
5.4.2 Create user profiles . . . . .	66
5.4.3 Location, location, location . . . . .	66
5.4.4 Secure the keystore . . . . .	67
5.4.5 Accessing the keystore . . . . .	68
5.4.6 Auditing keystore access . . . . .	69

<b>Chapter 6. Choosing a data encryption method</b> .....	71
6.1 Factors to consider .....	72
6.2 Choosing an interface .....	73
6.3 Choosing an algorithm .....	73
6.3.1 Cipher algorithm .....	73
6.3.2 Hash and HMAC algorithms .....	75
6.4 Tips and techniques .....	75
<b>Chapter 7. Database considerations</b> .....	77
7.1 Understanding how the database is used .....	78
7.2 How encryption impacts database structure .....	78
7.2.1 Modifying current record layout structure .....	80
7.2.2 Normalizing the encrypted fields .....	81
7.2.3 Key version field .....	82
7.3 Converting the plaintext data to ciphertext .....	83
7.3.1 Adjusting to database structure changes .....	84
7.3.2 Encrypting existing data .....	86
7.3.3 Reducing the initial data conversion window .....	87
7.3.4 Validating the encrypted data .....	89
7.4 Common tools for data maintenance and inquiry .....	90
7.4.1 AS/400 Data File Utility (DFU) .....	90
7.4.2 IBM Query for i5/OS (Query/400) .....	93
7.4.3 Interactive SQL .....	94
7.4.4 Other tools .....	94
<b>Chapter 8. Application considerations</b> .....	97
8.1 Accommodating database changes .....	98
8.1.1 Record format changes .....	98
8.1.2 Database normalization .....	98
8.2 Working with encrypted data .....	98
8.2.1 Performing encryption tasks with database triggers .....	99
8.2.2 Determining encryption state .....	100
8.2.3 Data sorting .....	103
8.2.4 Random access to encrypted data .....	106
8.2.5 Triggers .....	106
8.3 Other considerations .....	106
8.3.1 Spooled files .....	106
8.3.2 Exported data .....	107
<b>Chapter 9. Backup considerations</b> .....	109
9.1 Managing keys on a backup system .....	110
9.1.1 Coordinating keys between multiple systems .....	110
9.1.2 Translating keystores .....	110
9.1.3 Transporting keys between systems .....	110
9.2 Securing backup data .....	110
9.2.1 Transporting data to the backup system .....	110
9.2.2 Working with encrypted data between multiple systems .....	111
<b>Part 3. Implementation of data encryption</b> .....	113
<b>Chapter 10. SQL method</b> .....	115
10.1 Preparing for encryption .....	116
10.1.1 Encryption prerequisites .....	116
10.1.2 Identifying changes to your database .....	116

10.1.3 Analyzing impact to performance . . . . .	117
10.2 Encrypting data using an encryption password. . . . .	118
10.2.1 Associating a hint with a password. . . . .	118
10.2.2 Using a password in a view. . . . .	119
10.2.3 Using password and hint as encryption parameters. . . . .	119
10.3 Encrypting data with triggers. . . . .	120
10.3.1 Using classical triggers. . . . .	120
10.3.2 Using Instead Of Triggers. . . . .	120
10.4 Using user-defined functions (UDFs) with encrypted data . . . . .	121
10.5 Encrypting with stored procedures . . . . .	127
<b>Chapter 11. Cryptographic Services APIs method . . . . .</b>	<b>133</b>
11.1 Scenario description . . . . .	134
11.1.1 Setting up a master key . . . . .	134
11.1.2 Setting up a symmetric data encryption key . . . . .	135
11.1.3 Encrypting data . . . . .	136
11.1.4 Decrypting data . . . . .	137
11.1.5 Scenario analysis and summary of APIs used . . . . .	138
11.2 Scenario application setup . . . . .	139
11.2.1 Sample application download and initial setup . . . . .	139
11.2.2 Creating commands for sample application scenario . . . . .	139
11.3 Using the scenario application . . . . .	143
11.3.1 Create a master key: SET_MSTR_K command . . . . .	143
11.3.2 Create symmetric keys: GEN_SYMKEY command . . . . .	152
11.3.3 Encrypt data: SET_DATA command. . . . .	170
11.3.4 Decrypt data on source system: GET_DATA command. . . . .	185
11.3.5 Decrypt data on target system . . . . .	194
11.3.6 Execution example of scenario application. . . . .	198
11.4 Another scenario: for external UDFs functions . . . . .	202
11.4.1 External UDFs functions scenario overview . . . . .	204
11.4.2 HASH_DATA UDF function. . . . .	204
11.4.3 DEC_DATA UDF function. . . . .	207
11.4.4 Running DEC_DATA command . . . . .	209
11.4.5 Execution example of external UDFs function scenario . . . . .	213
11.4.6 Using the external trigger function . . . . .	218
<b>Chapter 12. HW-based method . . . . .</b>	<b>229</b>
12.1 Scenario overview. . . . .	230
12.1.1 Scenario A: exchanging secret data between two systems . . . . .	230
12.1.2 Scenario B: encryption/decryption of data on the same system . . . . .	232
12.2 Prerequisites and assumptions. . . . .	233
12.3 Scenario environment setup . . . . .	234
12.4 Exchanging secret data between two systems (scenario A). . . . .	235
12.4.1 Two systems scenario: step-by-step guide. . . . .	235
12.4.2 Execution example of scenario A . . . . .	262
12.5 Data encryption/decryption on same system (scenario B) . . . . .	268
12.5.1 Single system scenario: step-by-step guide . . . . .	268
12.5.2 Execution example of scenario B . . . . .	275
<b>Appendix A. Additional material . . . . .</b>	<b>281</b>
Locating the Web material . . . . .	281
Using the Web material . . . . .	281
System requirements for downloading the Web material . . . . .	281
How to use the Web material . . . . .	282



<b>Related publications</b> .....	283
IBM Redbooks .....	283
Other publications .....	283
Online resources .....	283
How to get Redbooks .....	283
Help from IBM .....	284
<b>Index</b> .....	285



# Notices

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

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.*

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

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

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

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

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

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


## COPYRIGHT LICENSE:

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

# Trademarks

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

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

AIX 5L™	i5/OS®	System i5™
AIX®	IBM®	System/38™
AS/400®	iSeries®	WebSphere®
DB2 Universal Database™	Redbooks®	z/OS®
DB2®	Redbooks (logo)  ®	
eServer™	System i™	

The following terms are trademarks of other companies:

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

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

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

# Preface

Regulatory and industry-specific requirements, such as SOX, Visa PCI, HIPAA, and so on, require that sensitive data must be stored securely and protected against unauthorized access or modifications. Several of the requirements state that data must be encrypted.

IBM® i5/OS® offers several options that allow customers to encrypt data in the database tables. However, encryption is not a trivial task. Careful planning is essential for successful implementation of data encryption project. In the worst case, you would not be able to retrieve clear text information from encrypted data.

This IBM Redbooks® publication is designed to help planners, implementers, and programmers by providing three key pieces of information:

- ▶ Part 1, “Introduction to data encryption” on page 1, introduces key concepts, terminology, algorithms, and key management. Understanding these is important to follow the rest of the book.

If you are already familiar with the general concepts of cryptography and the data encryption aspect of it, you may skip this part.

- ▶ Part 2, “Planning for data encryption” on page 37, provides critical information for planning a data encryption project on i5/OS.
- ▶ Part 3, “Implementation of data encryption” on page 113, provides various implementation scenarios with a step-by-step guide.

## The team that wrote this book

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

**Beth Hagemeister** is an IBM Advisory Software Engineer with over 20 years of experience developing cryptographic software for System/38™, AS/400®, and System i™ systems.

**John Concini** is a Senior Software Engineer at Vision Solutions, Inc., in Irvine, California. His 25 years of experience in software analysis and design includes development of compilers, communications software, and financial applications. Since 2002, he has contributed to high availability (HA) and disaster recovery (DR) solutions on the IBM System i platform.

**Milan Kalabis** is an IT Specialist working for IBM Czech Republic. He is an i5/OS Specialist and his expertise is in cryptography and communication.

**Robin Tatam** is a Senior System i Engineer for MSI Systems Integrators, an IBM Premier Business Partner in Des Moines, Iowa (USA). He has 18 years of experience in AS/400 and System i environments. Currently, Robin leads MSI’s System i security practice and regularly performs client training, vulnerability assessments, and security-related resolution services. Before joining MSI, Robin focused on commercial application development, including modernization with CGI and the IBM WebFacing Tool. His varied expertise has resulted in more than a dozen IBM certifications.

**Yessong Johng** is an IBM Certified IT Specialist at the IBM International Technical Support Organization, Rochester Center. He started his IT career at IBM as a S/38 Systems Engineer in 1982 and has been with S/38, AS/400, iSeries®, and System i now for 25 years. He writes

extensively and develops and teaches IBM classes worldwide on the areas of IT Optimization, and his topics include Linux®, AIX® 5L™, and Windows® implementations on the System i platform. His other coverage areas include TCP/IP, Data and Networking Security, and WebSphere®.

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

Rich Diedrich  
Terry Hennessy  
Kent Milligan  
Chad Sandbern  
Kevin Trisko  
IBM Rochester

Thomas Barlen  
IBM Germany

## Become a published author

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

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

Find out more about the residency program, browse the residency index, and apply online at:  
[ibm.com/redbooks/residencies.html](http://ibm.com/redbooks/residencies.html)

## Comments welcome

Your comments are important to us!

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

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

[ibm.com/redbooks](http://ibm.com/redbooks)

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

[redbooks@us.ibm.com](mailto:redbooks@us.ibm.com)

- ▶ Mail your comments to:

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



# Part 1

# Introduction to data encryption

This part introduces key concepts, terminology, algorithms, and key management. Understanding these is important in order to understand the rest of the book.







# Data encryption: the big picture

Traditionally, security has focused on establishing a perimeter of defense around system assets. While securing access points continues to be an important part of security, the typical business cannot afford to lock down its entire enterprise.

More and more, open networks are used to connect customers, partners, employees, suppliers, and their data. While this offers significant advantages, how does a business protect its information assets and comply with industry and legislative requirements for data privacy and accountability?

Encryption is *part* of the answer.

In this chapter, you will be introduced to some of the reasons that organizations are considering encryption technology. We also review the solid foundation that encryption requires in order to be deployed successfully.

## 1.1 Introducing encryption

If you pick up virtually any newspaper, or check any major news Web site, you will not have to wait long to see the next headline story of private data being exposed by either accidental or nefarious means. Whether it is a lost or stolen mobile computer, an intercepted electronic transmission, or a hacker breaching an organization's perimeter defenses, the numbers are staggering and do not show any sign of slowing.

Many organizations are now turning to encryption to protect their data, with almost an *expectation* that, at some point, their data will fall victim to the statistics.

### 1.1.1 What is data encryption

Encryption, or more properly *cryptography*, is the mathematical discipline, or the implementation of such, that transforms understandable information into unintelligible data, for the purpose of hiding and authenticating information.

Cryptographic functions on System i are used transparently, for example, with an i5/OS password, automatically in the configuration of technologies such as SSL, and manually in user applications.

By securing the data using cryptographic measures, information can travel throughout the network while maintaining data confidentiality and integrity, while user applications can also access cryptographic functions directly via application programming interfaces (APIs).

Using cryptographic functions can help provide the following:

- ▶ Data confidentiality (secrecy or privacy)  
Encryption is used to render data unintelligible. Only authorized entities are able to use decryption to make the data intelligible again.
- ▶ Data integrity  
Several cryptographic functions, such as digital signatures, message authentication codes, and keyed hashes, are used to help ensure data has not been altered.
- ▶ Authentication of communicating parties  
Digital signatures, message authentication codes, and some key agreement protocols are used to verify the origin of data.
- ▶ Non-repudiation  
A digital signature is used to prove the involvement of an entity in a previous action.

### 1.1.2 What drives the requirement

Organizations that consider encryption technologies are usually doing so for one of the reasons discussed below.

For more specific examples refer to "Regulations and standards" on page 40

#### **Government regulation**

Following a number of high-profile corporate scandals, many governments are stepping in to ensure that businesses conduct business in a safe and ethical manner.

In recent years, the U.S. Government and many states have enacted laws that directly impact the storage and availability of information, including the protection of private data.

Organizations of various sizes and in many business sectors are now required to pass audits to certify that they are compliant. Non-compliance can involve large fines and even jail time.

### **Industry rules**

As with government policy, industry policy can mean the difference in a business being able to operate and not being able to operate. For example, large companies that do not successfully reach the standards imposed by the major credit card firms can find themselves faced with significant fines and even the suspension of their ability to process card transactions—a feature that many retail organizations rely on to conduct business.

### **Business requirements**

With the frequency of data breaches, it is becoming a market advantage just to prevent being next in the list.

Organizations that operate in a competitive marketplace, such as the Internet, now not only have to promote their products and services, but also reassure the companies and individuals that purchase them, that when they provide their private information, it is going to be maintained securely. Similarly, companies that rely on proprietary information to separate themselves in the market, such as a unique formula or recipe, must protect that information or face losing their competitive edge.

Very large organizations often have the power to dictate to business suppliers that they either conform with published security policies, or they will find other suppliers. Often this is indicative of legal requirements that all links in their supply-chain be certified as secure. It is foreseeable that this may go as far as to requiring a supplier to pass a legislative audit to which they would not normally be subjected.

Last, but definitely not least, forward-thinking organizations realize that the requirement for encryption is only going to increase, and they should start planning for that infrastructure now.

## **1.1.3 Where to protect the data**

Data protection falls into two main categories, as explained below.

### **Data-in-motion**

Also known as data-in-flight, this term generically refers to protecting information any time the data leaves its primary location. For example, when data is transmitted from the database across any type of network (but typically *non-trusted* segments such as the public Internet), we look to use technologies such as Secure Sockets Layer (SSL), Virtual Private Networks (VPNs) and IP Security (IPSec) to assure data *confidentiality* (privacy), and other technologies such as digital certificates, message authentication codes, and keyed hashes, to ensure data *integrity*.

Data-in-motion also encompasses backup media, when that media leaves the data center—a core requirement for any valid disaster recovery strategy. We see policies that range from tapes sitting in employee cars until they are taken home (not recommended), to the contracted collection and storage by professional data storage firms, and both carry the significant risk of an unauthorized person coming into possession of your valuable information. During security assessments, we have even seen seemingly secure corporations that set tapes in public hallways awaiting courier pickup.

Electronic mail is another area, often overlooked as corporate data, that is still an open target for many organizations. It is relatively easy to intercept mail that may contain proprietary corporate details and confidential dialogue. It is also not difficult to alter that mail before forwarding it to the intended recipient.

All of these areas can be addressed with encryption-based technologies.

### **Data-at-rest**

Protecting data as it resides in the database is referred to as *at-rest*.

On System i, this protection entails utilization of the native i5/OS security controls, in conjunction with database encryption.

With the attention that is paid to securing data-in-motion, the database itself is often a much easier target to attack. By building layers of security over that database, you first increase the level of difficulty for an unauthorized user to even gain access to the data, and then compound that with the fact that private data is not stored in human-readable form.

If encryption is used as part of the strategy for the protection of data-at-rest, this also indirectly addresses the issue of exposed tape media as, even if tapes fall into the wrong hands, the data stored on them is unreadable without the correct key.

Of course, all of this assumes that you have enacted the appropriate key management techniques, as discussed in Chapter 5, “Managing keys on System i” on page 47.

## **1.2 Building on top of a secure foundation**

Before we discuss any encryption specifics, we need to point out that encryption is not an all-in-one solution to security.

Without an overall security strategy and careful consideration of *all* security aspects, encryption may cause more harm than good.

### **1.2.1 Only one piece of the security puzzle**

If your goal for encryption is to maintain data privacy during transmission, or to protect your backups when the media leaves your physical control, then basic i5/OS security-related controls, such as user profile attributes and system values, may seem to have little relevance.

However, before you embark on any encryption initiative, it is imperative to first take a good look at how solid your security environment is in its existing configuration. If you do not, then you run the risk of building a security infrastructure that crumbles under its own weight.

### **1.2.2 Security is not only about privacy**

Many people think of security as a synonym for privacy, and encryption as a vehicle for accomplishing it. However, this overlooks another significant reason that we need to establish a good security foundation. Our need for privacy is only truly satisfied if the information that we are protecting is accurate. Why bother encrypting information if its integrity cannot even be assured? If we engage in a project to encrypt your personal credit information, would you consider it to be truly secure if the data had been sitting in a database that everyone had unrestricted access to prior to the encryption? Probably not (and there is an entire industry selling consumer credit reports to prove it).

Security is also about maintaining the availability and integrity of your server as well as the information that resides on it, neither of which can be guaranteed without good security controls.

- ▶ If a user can alter the database at-will then the integrity of the data is in question.
- ▶ If a user has the ability to power down the server, and many unknowingly do so, then it does not even matter whether the data is encrypted, as it cannot be used for the legitimate business purpose for which we are storing it.
- ▶ If the user has the ability to alter the data *after* it is encrypted, then there is a good likelihood that the information can never be retrieved.

When faced with establishing security, we often focus on the tasks that we know how to perform, or address the exposures that we feel represent the greatest risk. But in reality, true security comes only when we address *all* of the risks.

- ▶ Why go to the trouble of locking your car if all of the windows are down?
- ▶ Why close the windows and lock the doors if everyone in the parking lot has a key?

It is only a matter of time before someone comes along with the desire to drive off with your car and you will be completely helpless to prevent them.

### 1.2.3 Surely not on System i

In their 2007 study, PowerTech (<http://www.powertech.com>), a System i security vendor, revealed that the average system that they audited had 82 profiles with All Object (\*ALLOBJ) special authority. In addition to numerous other capabilities, this authority provides a user with read/write access to every record in every file in every library.

This access comes regardless of any restrictions that you may have established for those files and, while it can be argued that encryption will still help mask private data from these individuals, that argument falls short when that individual realizes that they have the ability to access the keys to decrypt that data, as well as the ability to simply corrupt the ciphertext data so that it cannot be recovered.

Understand, however, that even a user *without* All Object special authority may have unrestricted access to your information if files are not properly secured at the object level. Object security controls have been included in the operating system since the servers inception, yet many shops still rely on trivial controls provided by their application or, worse yet, no controls at all.

### 1.2.4 Security initiatives to consider

Although the following list is not exhaustive, it provides suggestions for consideration. If you have not embarked on resolving the issues that pertain to your enterprise, then we strongly recommend that you start here:

- ▶ Establish a quality System i security policy to measure yourself against. Open source security policies exist if you do not wish to write your own.
- ▶ Hire an expert to perform a security analysis. Ensure that the person is experienced with security on System i, as this server is unique. Have them first review and then measure your System i against your security policy. If you believe that you are secure, this will serve to confirm and document that belief.
- ▶ Work to remove all special authorities from any user who does not have a demonstrable need for that administrative function. All Object is not the only capability that means

vulnerability on your system. For occasional access requirements, combine mechanisms such as command-line restrictions, profile switching, and profile auditing.

- ▶ Use authority controls provided in i5/OS to secure your objects and libraries. Employ profile switching and authority adoption techniques to enable data access through approved application interfaces.
- ▶ Establish system value settings based on your security policy requirements. Actively monitor for changes to these values.
- ▶ Control programmers using change management procedures and segregated test environments.
- ▶ Control and audit your client-initiated traffic (FTP, ODBC, and so on) using exit-point technology. You can write your own or buy them off the shelf.
- ▶ Audit administrative activities to security journals and follow policy on how that information is going to be reviewed.

## 1.2.5 A final word on traditional security

It is your responsibility to leverage the controls provided to you in i5/OS. For those that do, the System i has proven itself to be a formidable foundation, and encryption is a good building block to sit atop that. For those that do not, then encryption may not be the answer, as it cannot secure your data alone.

For more information about i5/OS security, refer to the following:

- ▶ *IBM System i Security Guide for IBM i5/OS Version 5 Release 4*, SG24-6668
- ▶ iSeries Security Reference Version 5, SC41-5302-06
- ▶ IBM Information Center, Version 5 Release 4 Web site  
<http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/rzahg/rzahgicsecurity.htm>

## 1.3 What this book is about

The focus of this book is on using cryptographic measures to secure data at rest. In other words, we examine how to protect data stored within a database file using cryptographic functions (specifically encryption, decryption, and authentication), as well as how to establish and maintain the keys that are critical to the availability and protection of that data.

We assume that you have taken heed of our warnings, and established a solid foundation via appropriate perimeter security and i5/OS object-level controls.

### 1.3.1 Book objectives

When you finish reading this book, our hope is that you will have a solid understanding of:

- ▶ Common cryptographic terms and concepts
- ▶ Available algorithms
- ▶ Key management techniques
- ▶ Database considerations
- ▶ Native application considerations
- ▶ SQL features

We walk you through each topic using a series of explanations as well as practical examples.

## 1.3.2 Book road map

This book is outlined below:

► Part 1, “Introduction to data encryption” on page 1

Read Part 1 if you have little to no knowledge of cryptography. If you do understand cryptography, you may still want to read 2.3, “System i cryptographic implementations overview” on page 22, for an overview of cryptography on System i.

– Chapter 1, “Data encryption: the big picture” on page 3

This is the current chapter that you are reading, which presented where encryption fits into the entire security structure.

– Chapter 2, “Algorithms, operations, and System i implementations” on page 11

This chapter introduces the novice to the various types of cryptographic algorithms and how they are combined to form different cryptographic operations. This chapter also outlines the System i cryptographic service providers and cryptographic interfaces.

– Chapter 3, “Key management concepts” on page 25

This chapter contains a general discussion on key types, key size, generating keys, storing keys, key separation, key distribution, key life-span, and key destruction.

► Part 2, “Planning for data encryption” on page 37

Read Part 2 to help you understand your requirements, the options that you have for encrypting data at rest, and how it will affect your database applications.

– Chapter 4, “Analyzing needs and defining scope” on page 39

This chapter introduces some of the common legislation pertaining to data security, and then outlines some topics about evaluating how encryption can be deployed in your organization.

– Chapter 5, “Managing keys on System i” on page 47

Chapter 5 introduces two System i key management interfaces and also discusses implementing your own.

– Chapter 6, “Choosing a data encryption method” on page 71

Chapter 6 provides guidance in choosing a cryptographic interface, a cryptographic algorithm, and mode of operation. It also provides some additional tips and techniques.

– Chapter 7, “Database considerations” on page 77

This chapter discusses the impact that hosting encrypted data has on i5/OS database design, how you move to an encrypted data model, and also how encrypting your data affects some popular utility applications commonly used to add, update, and view file data.

– Chapter 8, “Application considerations” on page 97

This chapter discusses the programming considerations for earlier applications when working with data stored in encrypted form.

– Chapter 9, “Backup considerations” on page 109


This chapter discusses backing up keys and the data that they encrypt.

► Part 3, “Implementation of data encryption” on page 113

Part 3 provides examples and discussions of three methods for encrypting data—SQL, Cryptographic Services APIs, and using the Cryptographic Coprocessors.

- Chapter 10, “SQL method” on page 115  
This chapter demonstrates how SQL provides a simple and convenient method for encrypting and decrypting data. It also discusses how SQL does not provide key management, and provides sample functions to assist with the management of passwords and keys.
- Chapter 11, “Cryptographic Services APIs method” on page 133  
This chapter demonstrates use of the Cryptographic Services APIs, which primarily use software-based encryption and optionally the 2058 Cryptographic Accelerator.
- Chapter 12, “HW-based method” on page 229  
In this chapter, we demonstrate use of the Cryptographic Coprocessor 4764/4758 for secure hardware-based cryptography. The Common Cryptographic Architecture (CCA) API set is used to interface with the Cryptographic Coprocessor 4764/4758 hardware.





## Algorithms, operations, and System i implementations

Different cryptographic *operations* may use one or more *algorithms*. You choose the cryptographic operation and algorithms depending on your purpose. For example, for the purpose of ensuring data integrity, you might want to use a message authentication code (MAC) operation with the Advanced Encryption Standard (AES) algorithm.

System i supports many cryptographic algorithms and operations via four programming interfaces.

## 2.1 Cryptographic algorithms

A *cryptographic algorithm* is a mathematical procedure that is used in the transformation of data for the purpose of securing the data.

### 2.1.1 Cipher algorithms

In this book, when we speak of a *cipher* algorithm, we are referring to a cryptographic algorithm that can transform understandable information (*plaintext*) into an unintelligible piece of data (*ciphertext*), and can transform that unintelligible data back into understandable information.

**Note:** There are many ways of rendering data unreadable, and you might be tempted to write your own. You should not. The development of cryptographic algorithms is the work of highly trained cryptographers. Even then, the algorithms are submitted to extreme scrutiny by the security community, and the algorithms are often found lacking. The algorithms addressed in this book have been approved by the security community and most have become standards.

The cipher algorithms supported by i5/OS are all in the public domain. Therefore, the algorithm itself is not secret. So what protects the data? The cipher algorithm requires a *key* as input. It is the key that instructs the algorithm how to *encrypt* (scramble) and *decrypt* (unscramble) the data. It cannot be emphasized enough that the security of the encrypted data depends on safeguarding your key.

There are two types of cipher algorithms: symmetric and asymmetric.

#### Symmetric algorithms

With a *symmetric* or *secret key* algorithm, the key is a shared secret between two communicating parties. Encryption and decryption both use the same key.

In Figure 2-1, Alice and Bob have previously exchanged a key via a secure method. (Methods of key distribution are discussed in 3.8, “Key distribution” on page 34.) When Alice encrypts her message, she must supply the key to the encryption routine. When Bob receives the encrypted message, he also must supply the key to decrypt the message. Others may intercept the message, but without the key, it is unreadable.

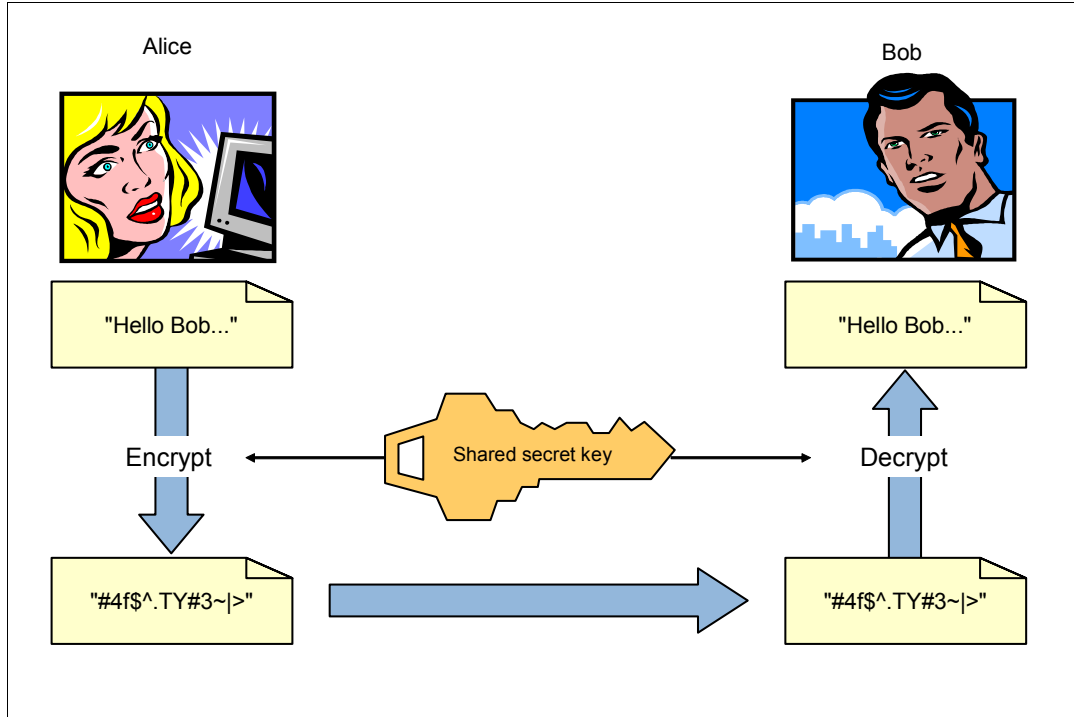


Figure 2-1 Symmetric cipher

There are two types of symmetric key algorithms:

- ▶ Block cipher

In a block cipher, the cipher algorithm works on a fixed-size block of data. For example, if the block size is eight, eight bytes of plaintext are encrypted at a time. Normally, the user's interface to the encrypt/decrypt operation handles data longer than the block size by repeatedly calling the low-level cipher function.

- ▶ Stream cipher

Stream ciphers do not work on a block basis, but convert 1 bit (or 1 byte) of data at a time. Basically, a stream cipher generates a keystream based on the provided key. The generated keystream is then XORed with the plaintext data.

### **Block ciphers**

The following block ciphers are supported by i5/OS.

- ▶ DES

Data Encryption Standard (DES), also known as Data Encryption Algorithm (DEA), was adopted as a Federal Information Processing Standard (FIPS) in 1976. It is described in *FIPS Pub 46-3, Data Encryption Standard (DES)*, at the following URL:

<http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>

Although DES is still widely used, it is no longer considered secure due to its short key length. The FIPS standard was withdrawn in 2005. You should not use DES except for compatibility purposes.

- ▶ TDES

Triple DES (TDES), also known as Triple Data Encryption Algorithm (TDEA), was the natural successor to DES. TDES uses the DES algorithm three times with either a double-length or a triple-length DES key. TDES operates on a block of plaintext by doing a DES encrypt, followed by a DES decrypt, and then another DES encrypt. Therefore, it uses three DES keys. If only two DES keys are supplied (called 2TDES), the first key is used for both encrypt operations. (When three keys are supplied, it is called 3TDES.) The National Institute of Standards and Technology (NIST) defines TDES in *NIST Special Publication 800-67, Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher* at the following URL:

<http://csrc.nist.gov/publications/nistpubs/800-67/SP800-67.pdf>

- ▶ AES

Advanced Encryption Standard (AES) was adopted by NIST after an international competition in 2001 and is now the official replacement to DES. Because the performance of AES is better than TDES, it has gained widespread use. AES is documented in *FIPS Pub 197, Advanced Encryption Standard (AES)*, at the following Web site:

<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>

- ▶ RC2

Rivest Cipher 2 (RC2) was invented by Ron Rivest, co-founder of RSA Data Security, in the 1980s. RC2 has a variable key size. It was designed as a replacement for DES with special export privileges for a small key size. Because of this and its performance in comparison to DES, it gained widespread use. RC2 is described in *RFC 2268, A Description of the RC2 Encryption Algorithm* at the following Web site:

<http://www.ietf.org/rfc/rfc2268.txt>

Because block ciphers encrypt a block of data at a time, two problems present themselves:

- ▶ Patterns can be left in the encrypted data. Consider how the start and end of messages will often repeat. When using the same key, the same block of plaintext will always encrypt to the same ciphertext. Cryptanalysts can use this information to mount attacks.
- ▶ The plaintext data must be a multiple of the block size. If not, the data must be padded before encrypting.

*Modes of operation* are used to deal with both these problems. A mode of operation defines a means to add feedback into the encryption operation to mask patterns in the data. Some modes of operation also deal with plaintext data whose length is not on a block boundary. More information about modes of operation can be found in *NIST Special Publication 800-38A, Recommendation for Block Cipher Modes of Operation* at the following URL:

<http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>

► CBC

With Cipher Block Chaining (CBC) mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This will hide patterns in the data. CBC mode is depicted in Figure 2-2.

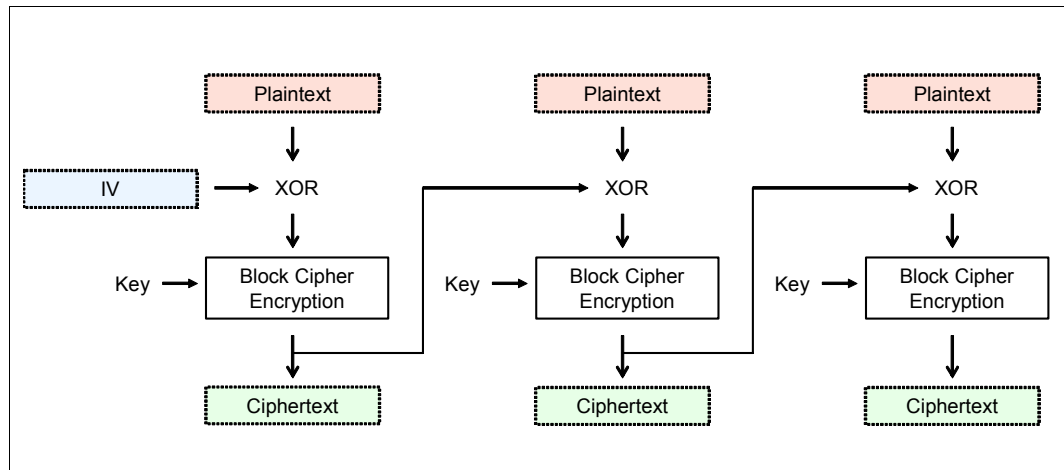


Figure 2-2 Cipher Block Chaining

Note that CBC mode requires an *initialization vector* (IV). Because there is no previous ciphertext to XOR into the first block of plaintext, an IV must be supplied. Using an IV is described in more detail in Chapter 6, “Choosing a data encryption method” on page 71.

CBC is usually your best choice of mode.

► CUSP

Cryptographic Unit Support Program (CUSP) is a special type of CBC mode documented in the *z/OS ICSF Application Programmer's Guide, SA22-7522*. It is used for handling data not a multiple of the block length. V5R4 i5/OS does not support CUSP mode of operation, but an application can implement it. For details refer to “AES” on page 74.

► ECB

Electronic Code Book (ECB) is also a mode of operation, but without any feedback mechanism applied. Therefore, it does not solve either of the problems above. ECB should only be used when encrypting random data, such as keys.

**Stream ciphers**

The Rivest Cipher 4 (RC4) stream cipher is supported by i5/OS. RC4 was invented by Ron Rivest in 1987. It is probably the most widely used stream cipher. It is used by Secure Sockets Layer (SSL) to protect Internet traffic. RC4 is an extremely efficient algorithm, and like all stream ciphers, the ciphertext will always be the same length as the plaintext. The disadvantage of using RC4 is that it is difficult to get the security correct. Specifically, key generation must be done correctly, or security will be severely compromised.

**Note:** Certain modes of operation can turn block ciphers into stream ciphers:

- ▶ Cipher Feedback (CFB) mode
- ▶ Output Feedback (OFB) mode

## Asymmetric algorithms

One of the difficulties of symmetric key algorithms is exchanging the key. How do Alice and Bob exchange their shared secret key in a secure manner?

With an *asymmetric* or *public key* algorithm (*PKA*), a pair of keys is used. One of the keys, the *private key*, is kept secret and is not shared with anyone. The other key, the *public key*, is not secret and can be shared with anyone. When data is encrypted by one of the keys, it can only be decrypted and recovered by using the other key. The two keys are mathematically related, but it is virtually impossible to derive the private key from the public key.

In Figure 2-3, Bob has generated a private/public key pair and has sent his public key to Alice. Alice encrypts her message with Bob's public key. Bob is the only party that can decrypt the message because only he has possession of the corresponding private key.

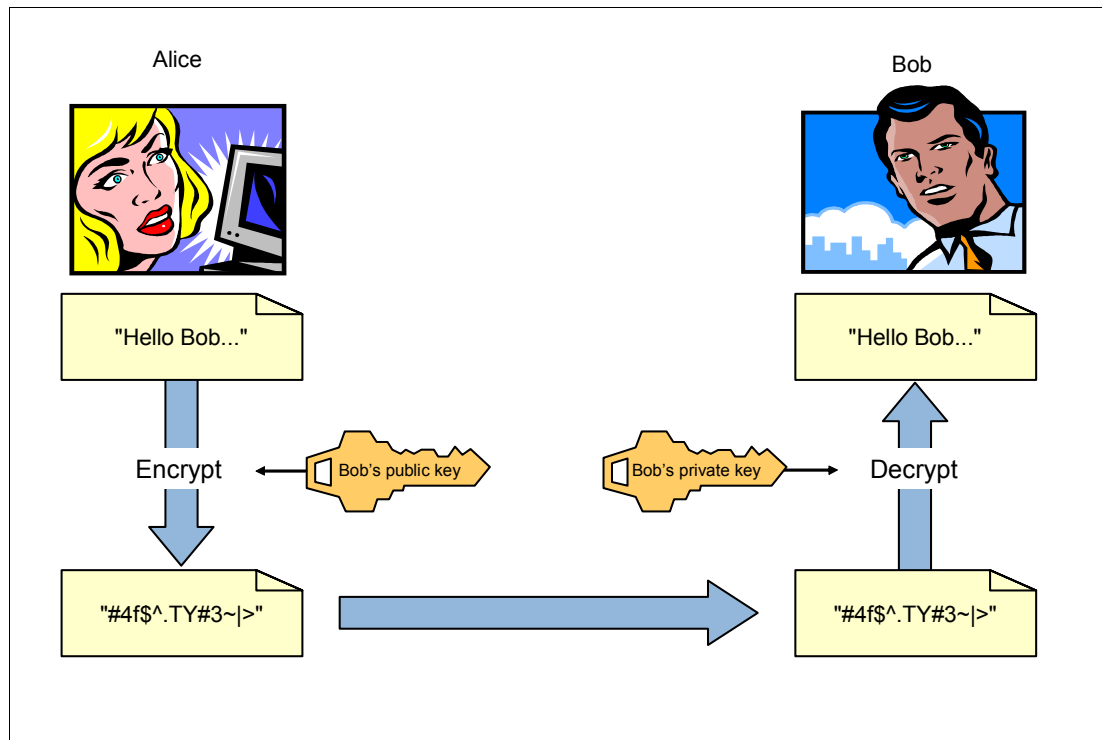


Figure 2-3 Asymmetric cipher

The advantage of using asymmetric keys is that the public key can be distributed freely without any security exposures. Data encrypted with the public key can only be decrypted by the party owning the private key.

The disadvantage of asymmetric algorithms is slow performance. Consequently, applications typically use asymmetric algorithms to distribute symmetric keys, and then do their bulk encryption using a symmetric key algorithm. So, in actuality, Alice would not encrypt her message with Bob's public key, but encrypt a symmetric key with the public key and send that to Bob. Then she would encrypt her message with the symmetric key and send that as depicted in Figure 2-1 on page 13.

*Rivest, Shamir, Adleman (RSA)* is an asymmetric algorithm supported by i5/OS that can be used to encrypt and decrypt. It is named after its inventors who first published it in 1977. Besides using RSA to encrypt keys for secure distribution, RSA can be used to encrypt digests used in the generation of digital signatures. This is described in “Sign/verify” on page 21.

## 2.1.2 Key distribution algorithms

When encrypted data must be decrypted at another location, distributing the key in a secure manner can be a challenge. Methods of key distribution are discussed in more detail in 3.8, “Key distribution” on page 34. The algorithms are:

- ▶ RSA

As previously discussed, RSA is a PKA algorithm. It is often used to distribute symmetric keys. Bob generates a public/private RSA key pair and sends the public key in the clear to Alice. Alice generates a symmetric key, encrypts it with Bob’s public key, and sends that to Bob. Bob decrypts the symmetric key using his private key.

- ▶ Diffie-Hellman

Diffie-Hellman is a PKA algorithm invented by Whitfield Diffie and Martin Hellman in 1976 for establishing a shared secret key across an insecure communications channel. It is described in *Public Key Cryptography Standard (PKCS) #3, Diffie-Hellman Key Agreement Standard* and in *RFC 2631, Diffie-Hellman Key Agreement Method*, found at the following URL:

<http://www.ietf.org/rfc/rfc2631.txt>

Basically, Bob generates a set of PKA parameters. These parameters are not secret and are sent to Alice in the clear. Both Alice and Bob use the parameters to generate a private/public key pair and send their public key to each other. Both Alice and Bob calculate a shared secret key using their own private key and the other’s public key. Note that Diffie-Hellman cannot be used for encryption/decryption purposes.

## 2.1.3 One-way hash algorithms

A *one-way hash* algorithm produces a fixed-length output string (often called a *digest*) from a variable-length input string. For all practical purposes, the following statements are true of a good hash function:

- ▶ Collision resistant

If any portion of the data is modified, a different hash will be generated.

- ▶ One-way

The function is irreversible. That is, given a digest, it is not possible to find the data that produces it.

i5/OS supports several hash algorithms:

- ▶ MD5

Message Digest 5 (MD5) was invented by Ron Rivest in 1991. It is described in *RFC 1321, The MD5 Message-Digest Algorithm*, found at the following URL:

<http://www.ietf.org/rfc/rfc1321.txt>

MD5 produces a 128-bit (16-byte) digest. MD5 has been widely used. However, serious flaws have been found in the algorithm, and we no longer recommend it for use.

► SHA-1

Secure Hash Algorithm 1 (SHA-1) was developed by the National Security Agency (NSA) and is published in *FIPS Pub 180-2, Secure Hash Standard*, with all the other SHA variations, at the following URL:

<http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf>

SHA-1 produces a 160-bit (20-byte) digest. The security of SHA-1 was recently compromised. We recommend using an SHA-2 algorithm if possible.

► SHA-2

Secure Hash Algorithm 2 (SHA-2) is a set of additional SHA algorithms published by NIST to match the higher security levels of longer keys:

- SHA-256 produces a 256-bit (32-byte) digest.
- SHA-384 produces a 384-bit (48-byte) digest.
- SHA-512 produces a 512-bit (64-byte) digest.

► MDC

Modification Detection Code (MDC) is a one-way hash invented by IBM that uses DES encryption and a default key. MDC produces a 128-bit (16-byte) digest.

► RIPEMD-160

RACE Integrity Primitives Evaluation Message Digest (RIPEMD-160) was developed in the European academic community. It was first published in 1996. RIPEMD-160 produces a 160-bit (20-byte) digest.

## 2.1.4 Random number generation algorithms

Many security-related functions rely on random number generation.

Random number generation is performed both in i5/OS via Cryptographic Services and on the cryptographic coprocessors via CCA. Both use a FIPS-approved pseudorandom number generator (PRNG).

**Note:** Random number generation functions implemented in software cannot be truly random because the process is entirely deterministic. Hence the term *pseudorandom* number generator. Usually, PRNGs are initialized with unpredictable data, called seed data. Seed data may be supplied by a user, generated automatically from the state of the computer, or obtained from a hardware true random number generator. *Cryptographically strong* PRNGs use cryptographic functions. Along with good seed data, they produce unpredictable and statistically random numbers.

On the cryptographic coprocessor, an electronic noise source provides unpredictable input to a random bit-value accumulator. Periodically, the hardware outputs seed to a FIPS 140-1 approved pseudorandom number generator.

The i5/OS pseudorandom number generator resides in the System i Licensed Internal Code (LIC). It uses a PRNG algorithm from Appendix 3 of *FIPS 186-2, Digital Signature Standard (DSS)*, found at the following Web site:

<http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf>

Cryptographically strong pseudorandom numbers rely on good seed. The FIPS 186-1 algorithm is seeded from a system seed digest. The system automatically generates seed using data collected from system information or by using the random number generator



function on a cryptographic coprocessor if one is available. System-generated seed can never be truly unpredictable. If a cryptographic coprocessor is not available, you should add your own random seed (via the Add Seed for Pseudorandom Number Generator API) to the system seed digest. This should be done as soon as possible any time the Licensed Internal Code is installed.

## 2.1.5 Summary of algorithms

Figure 2-4 summarizes the System i supported cryptographic algorithms.

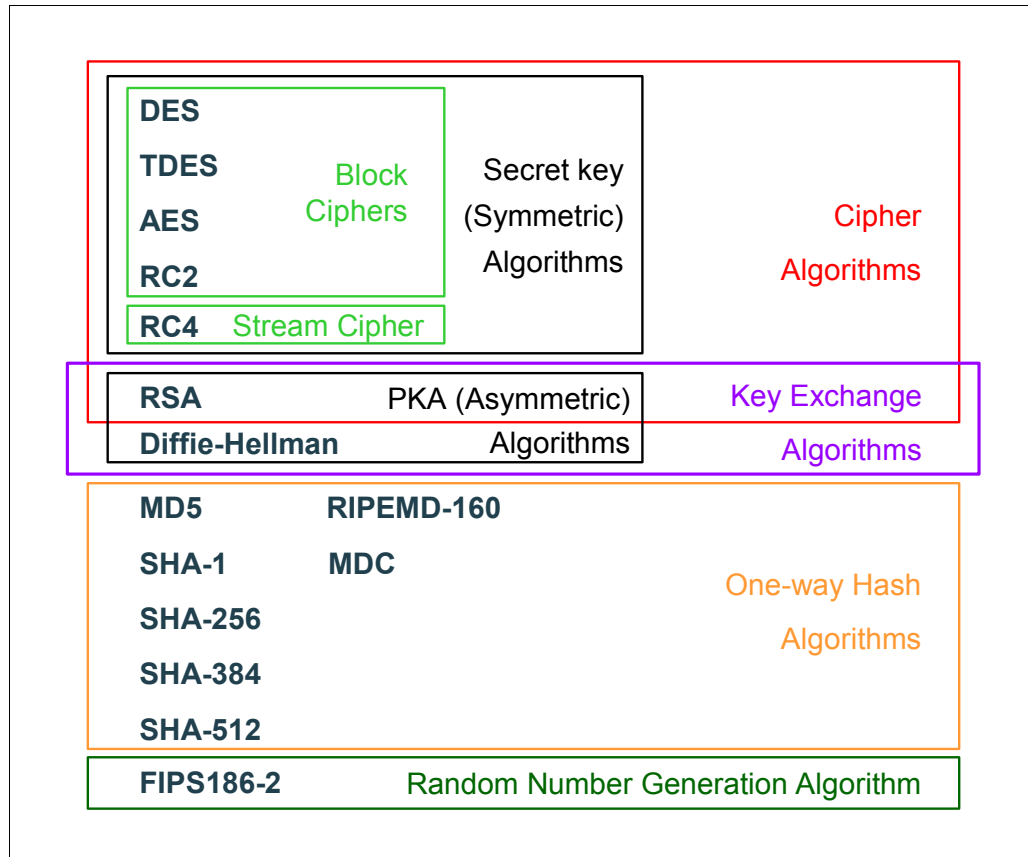


Figure 2-4 System i cryptographic algorithms

**Note:** Chapter 6, “Choosing a data encryption method” on page 71, provides more information about choosing an appropriate cipher algorithm and mode of operation.

## 2.2 Cryptographic operations

The algorithms discussed in the previous section are used individually or in combination in different cryptographic operations. We can roughly categorize cryptographic operations into five groups: data confidentiality, authentication (including data integrity and non-repudiation), random number generation, financial PINs, and key management.

## 2.2.1 Data confidentiality

Cryptographic operations for the purpose of data confidentiality prevent an unauthorized person from reading a message. The following operations are included in data confidentiality.

### Encrypt and decrypt

The encrypt operation changes plaintext data into ciphertext through the use of a cipher algorithm and key. To restore the plaintext data, the decrypt operation must employ the same algorithm and key.

Encryption and decryption may be employed at any level of the operating system. Basically, there are three levels:

- ▶ Link level encryption

Link level encryption is performed at the lowest level of the protocol stack, usually by specialized hardware.

- ▶ Session level encryption

With encryption at the session layer, the system requests cryptographic services instead of an application. The application may or may not be aware that encryption is happening.

- ▶ Field level encryption

With field level encryption, the user application explicitly requests cryptographic services. The user application completely controls key generation, selection, distribution, and what data to encrypt.

**Note:** This book deals with field level encryption.

### Translate

The translate operation decrypts data from encryption under one key and encrypts the data under another key. This is done in one step to avoid exposing the plaintext data within the application program.

## 2.2.2 Data authentication, integrity, and non-repudiation

Encrypted data does not mean that the data cannot be manipulated (for example, repeated, deleted, or even altered). To rely on data, you need to know that it comes from an authorized source and is unchanged. Additional cryptographic operations are required for these purposes.

### Hash (or message digest)

Hash operations are useful for authentication purposes. For example, you can keep a copy of a digest for the purpose of comparing it with a newly generated digest at a later date. If the digests are identical, the data has not been altered.

### Message authentication code (MAC)

A MAC operation uses a secret key and symmetric cipher algorithm. The input data is encrypted using CBC mode. But instead of returning the entire ciphertext, it returns the last block of encrypted data. This value is called a MAC and is used to ensure that the data has not been modified. Typically, a MAC is appended to the end of a message. The receiver of the message uses the same MAC key and algorithm to reproduce the MAC. If the receiver's MAC matches the MAC kept with the message, the data has not been altered.

The MAC operation helps authenticate messages, but does not prevent unauthorized reading because the data remains as plaintext. You must use the MAC operation and then encrypt the entire message to ensure both data privacy and integrity.

### **HMAC (hash MAC)**

An HMAC, or keyed hash, operation uses a cryptographic hash function and a shared secret key to produce an authentication value. It is used in the same way in which a MAC is used. Because hash algorithms are faster than symmetric ciphers, producing a MAC via an HMAC operation will perform better than producing a MAC via a symmetric cipher.

### **Sign/verify**

A sign operation produces an authentication value called a digital signature. A sign operation works as follows:

1. The data to be signed is hashed to produce a digest.
2. The digest is encrypted using a PKA algorithm such as RSA and a private key, to produce the signature.

The verify operation works as follows:

1. The signature is decrypted using the sender's PKA public key to produce digest 1.
2. The sender's data is hashed to produce digest 2.
3. If the two digests are equal, the signature is valid.

Theoretically, this also verifies the sender because only the sender should possess the private key. However, how can the receiver verify that the public key actually belongs to the sender? Certificates are used to help solve this problem.

## **2.2.3 Key and random number generation**

Many security-related functions rely on random number generation, for example, salting a password or generating an initialization vector. An important use of random numbers is in the generation of cryptographic key material. Key generation has been described as the most sensitive of all computer security functions. If the random numbers are not cryptographically strong, the function will be subject to attack.

## **2.2.4 Financial PINs**

Although not covered in this book, personal identification number (PIN) generation and handling are also considered cryptographic operations.

A PIN is a unique number assigned to an individual by an organization. PINs are commonly assigned to customers by financial institutions. The PIN is typed in at a keypad and compared with other customer-associated data to provide proof of identity.

To generate a PIN, customer validation data is encrypted with a PIN key. Other processing is done on the PIN as well, such as putting it in a particular format.

The cryptographic coprocessors provide a complete set of financial PIN operations. Cryptographic Services does not support any type of PIN operation.

## 2.2.5 Key management

Key management is the secure generation, handling, and storage of cryptographic keys. This includes key storage and retrieval, key encryption and conversions, and key distribution. Key management is discussed in more detail in Chapter 3, “Key management concepts” on page 25.

## 2.3 System i cryptographic implementations overview

This section provides an overview how cryptographic functions are implemented on System i.

### 2.3.1 Cryptographic service providers

System i supports several cryptographic service providers (CSPs). A CSP is the software or hardware that implements a set of cryptographic algorithms and operations.

- ▶ 4758/4764 Cryptographic Coprocessors

IBM 4764 Cryptographic Coprocessor is available on System i5™ and eServer™ i5 models as hardware feature code 4806. IBM 4758 Cryptographic Coprocessor is no longer available, but is still supported.

The coprocessors were designed to meet Federal Information Processing Standard (FIPS) PUB 140 level 4 security requirements. Information about FIPS 140 can be found at the following Web site:

<http://csrc.nist.gov/cryptval/cmvp.htm>

The Cryptographic Coprocessors contain a tamper-responding hardware security module (HSM) encapsulating a general-purpose processor, specialized cryptographic electronics, and non-volatile key storage. The coprocessors support an access-control system that is separate from i5/OS access controls.

More information about the Cryptographic Coprocessors can be found in the i5/OS Information Center, Version 5 Release 4, at the following URL:

<http://publib.boulder.ibm.com/infocenter/iseri5/v5r4/topic/rzajc/rzajcco4758.htm>

- ▶ 2058 Cryptographic Accelerator

The 2058 Cryptographic Accelerator is no longer available, but is still supported. The Cryptographic Accelerator was designed to improve the performance of SSL for those applications that do not require hardware-secured key storage. In addition, it can be used by Cryptographic Services APIs to offload processing. More information about the 2058 Cryptographic Accelerator can be found in the i5/OS Information Center, Version 5 Release 4, at the following URL:

<http://publib.boulder.ibm.com/infocenter/iseri5/v5r4/topic/rzajc/rzajccacel2058.htm>

- ▶ i5/OS LIC

A number of cryptographic algorithms are implemented within i5/OS Licensed Internal Code. These algorithms are used by many system functions and are available for application use through the Cryptographic Services API set.

- ▶ Java™ Cryptography Extensions

Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code algorithms. JCE is a standard extension to the Java 2 Software Development Kit (J2SDK),

Standard Edition. More information about IBM JCE can be found in the i5/OS Information Center, Version 5 Release 4, at the following Web site:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/topic/rzaha/rzahajce.htm>

## 2.3.2 Cryptographic interfaces

The following cryptographic API sets are available to System i applications for encrypting data at rest. Although this book only discusses the first three options, JCE is included for completeness.

### ► CCA

The Common Cryptographic Architecture (CCA) API set is provided for running cryptographic operations on a Cryptographic Coprocessor. The CCA and cryptographic coprocessors are used by System i for a number of purposes:

- By applications for general-purpose cryptographic operations.
- By applications for financial operations, such as PIN transactions, bank-to-clearing-house transactions, Europay, MasterCard, VISA (EMV) transactions for integrated circuit (chip) based credit cards, and basic Secure Electronic Transaction (SET) block processing.
- By SSL to offload intense cryptographic processing.
- By Digital Certificate Manager (DCM) to generate and securely store RSA keys, including private keys for SSL certificates.
- By the system pseudorandom number generator to obtain real random data for seed.

The CCA API set is described at the following Web site:

<http://www-03.ibm.com/security/cryptocards/library.shtml>

### ► i5/OS Cryptographic Services

The i5/OS Cryptographic Services API set is provided for running general-purpose cryptographic operations within the Licensed Internal Code or optionally on the 2058 Cryptographic Accelerator. The APIs are documented in the i5/OS Information Center, Version 5 Release 4, at the following URL:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/topic/apis/catcrypt.htm>

### ► SQL

Structured Query Language (SQL) supports encryption/decryption of database fields. The SQL built-ins for encrypting and decrypting DB2® data are described in the i5/OS Information Center, Version 5 Release 4, at the following URL:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/topic/db2/rbafzmstch2func.htm>

### ► Java Cryptography

The JCE implementation on System i is compatible with the implementation of Sun™ Microsystems, Inc., documented at the following URL:

<http://java.sun.com/products/jce/index.jsp>

Refer to 6.2, “Choosing an interface” on page 73, for a comparison of these interfaces.

Table 2-1 indicates what CSP is used under each interface.

Table 2-1 System i cryptographic service providers and interfaces

APIs/CSPs	i5/OS LIC	4764/4758	2058	JCE
CCA		X		
Cryptographic Services	X		X	
SQL	X			
Java Cryptography				X

Figure 2-5 depicts the System i implementation of cryptographic interfaces and service providers.

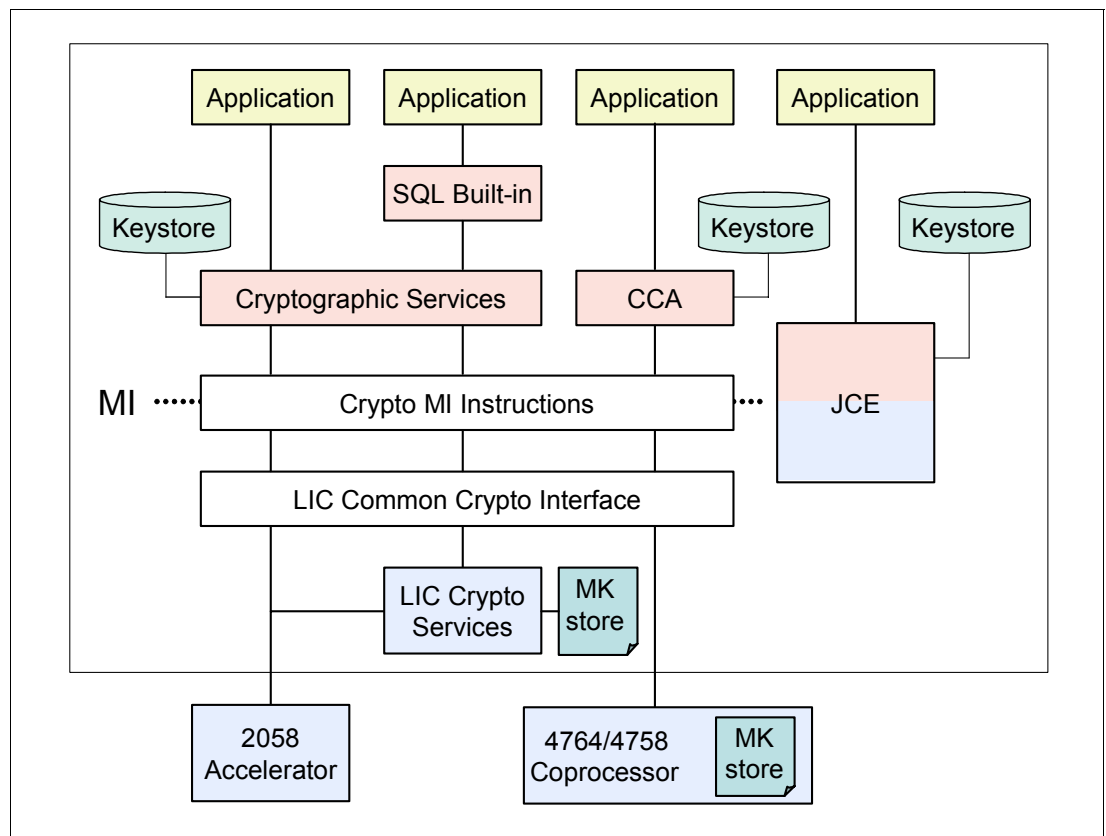


Figure 2-5 System i cryptographic implementation



## Key management concepts

How secure is your encrypted data? It depends on the security of your keys. The security of your keys depends on many factors, such as how you generate keys, store keys, authorize keys, distribute keys, and destroy keys.

## 3.1 Key management considerations

An encryption algorithm requires a key to transform the data. All cryptographic algorithms, at least the reputable ones, are in the public domain. Therefore, it is the key that controls access to the data. We cannot emphasize enough that you *must* safeguard the key to protect the data.

All of the following affect the security of your keys and therefore your encrypted data:

- ▶ The type and size of key
- ▶ How the keys are generated
- ▶ Where the keys are stored
- ▶ Access to keys
- ▶ How applications handle keys
- ▶ Key distribution
- ▶ If and how keys are backed up
- ▶ When keys are destroyed

## 3.2 Size matters

An obvious technique for hacking data encrypted with a symmetric key is an exhaustive search of all possible key values. Therefore, the larger you make your key size, the more prohibitive the search becomes.

We discuss key size, or the strength of keys, in terms of number of bits. However, on some interfaces you may be required to specify the key size in bytes, not bits. For example, to specify a 256-bit key size, you may be required to specify 32 bytes.

Deciding on a key size is an important decision. While a larger key size better protects the data, it can also decrease performance. Other factors to consider when deciding on a key size are:

- ▶ What key sizes are supported by the algorithm that you want to use?

Some algorithms only support a few key sizes. Others support a large range of key sizes. Table 3-1 lists i5/OS-supported key sizes.

Table 3-1 i5/OS V5R4 supported key sizes

	Cryptographic Services	Cryptographic Coprocessors	SQL built-in function
DES <sup>a</sup>	56	56	_b
Triple DES <sup>c</sup>	56, 112, 168	56, 112, (168) <sup>d</sup>	112
AES	128, 192, 256	-	-
RC2	8–1024 <sup>e</sup>	-	128
RC4-compatible	8–2048 <sup>f</sup>	-	-
RSA	512–2048 <sup>g</sup>	1024, 1280, 1536, 1792, 2048	-
Diffie-Hellman	512–1024 <sup>h</sup>	-	-

a. DES keys are always 8 bytes in length. So why is the key size not 64? One bit of each byte is sometimes used for parity and is not actually involved in the encryption algorithm. Therefore, the strength of the key is actually 56.



- b. “-” indicates that the algorithm is not supported.
- c. Like DES, one bit from each byte of a triple DES key is used for parity.
- d. CCA uses 168-bit triple DES keys for master keys only.
- e. Key size must be a multiple of 8.
- f. Key size must be a multiple of 8.
- g. The key size must be an even number.
- h. The key size must be a multiple of 64.

► What is your desired minimum security level?

By security level, we mean the number of steps that it takes to mount a successful attack. For example, in the case of an exhaustive search on a 56-bit value, the number of steps is  $2^{56}$ .

All your cryptographic operations should be designed to meet the security level that you establish. It does not make sense to encrypt data using a 256-bit key, and then encrypt the data-encrypting key under a 128-bit key.

Because many encryption modes are susceptible to collision attacks, for a security level of  $n$  bits, you should use a key at least  $2n$  bits. For example, for 128-bit security, you should use a 256-bit key.

► How long will the encrypted data be around?

This is an important consideration because the capabilities for exhaustive attacks in the future will be significantly better than today. If you plan to encrypt and store data for a long period of time (without translating at a future date) you need to choose a large key size. Cryptographic experts argue that systems today should aim at securing data for 30 to 50 years. That requires designing a system with a 128-bit security level.

Comparing the strength of keys can be tricky. While you can compare the number of bits for various symmetric keys (for example, triple DES, AES, RC2, RC4), you cannot compare number of bits between symmetric and asymmetric keys. In fact, you cannot always compare number of bits between various PKA algorithms.

Table 3-2 shows the PKA modulus sizes recommended by NIST for the RSA and Diffie-Hellman algorithms to meet various security levels.<sup>1</sup>

**Note:** A modulus is the product of two large primes used in modulo operations by the PKA algorithm. The security of the algorithm depends on the difficulty of factoring the modulus. Consequently, increasing the modulus size increases the security, but it also becomes much more CPU intensive.

Table 3-2 NIST recommendations for PKA key strength

For security level	Use PKA modulus
80	1024
112	2048
128	3072
192	7680
256	15360

The cryptographic service providers for i5/OS do not support the larger PKA key sizes. (The performance for large RSA keys is prohibitive.) Hopefully, in the future i5/OS will support

newer and stronger PKA technologies, such as Elliptic Curve Cryptography (ECC). Until then, you should use at least a 2,048-bit PKA key size.

## 3.3 Establishing a key value

When you establish a key on i5/OS, you will either receive the key value from another party, or generate one yourself.

### 3.3.1 Generating a key value

A large key size does no good if someone can easily guess the key value. For example, you should not use words for a key value. A program can quickly do an exhaustive search of all words in a dictionary. Using words for a key value provides very little unpredictable data, or *entropy*.

Sometimes words are used to set a key by entering a password or passphrase that is then hashed to the appropriate key size by the system (such as with the Set Encryption Password SQL function). In these situations, a large amount of character data must be entered to obtain good entropy. A good rule of thumb is to assume that your passphrase is only providing 1–2 bits of entropy per byte. So, for example, if you plan to use a 128-bit key you want 128 bits of entropy, and therefore it would be best to enter a passphrase containing 128 characters.

To generate a random key value, and therefore maximum entropy, use a key generation function or a cryptographically strong random or pseudorandom number generator.

When you establish important keys, such as master keys, it is a good idea to separate the key into parts so that no one person knows the key value. Full-function cryptographic service providers (CSPs) provide a means of multi-part master key establishment. However, to do so for other keys, you may need to write a program that first collects the key parts and then uses the CSP interfaces to establish the key.

### 3.3.2 Using a known key value

Sometimes you will be given a key value. For example, you may be using a key distribution protocol and the key arrives at your system encrypted with another key. Most likely, you will want to add this key into your keystore. A key import operation is used to accomplish this. A key import operation translates the key by decrypting it from the key-encrypting key and encrypting it under the master key. It then optionally stores the key in keystore. This operation is done without ever exposing the key in cleartext within your application. This should be a goal in your key management, to reduce exposure of clear key values as much as possible.

---

<sup>1</sup> SP 800-57 Part 1, *Recommendation for Key Management - Part 1: General (Revised)*, found at: [http://csrc.nist.gov/CryptoToolkit/kms/SP800-57Part1\\_3-8-07.pdf](http://csrc.nist.gov/CryptoToolkit/kms/SP800-57Part1_3-8-07.pdf)

## 3.4 Storing keys

There are many places keys can reside—some transient, some permanent. Basically, keys can be stored wherever data can be stored:

- ▶ As a program constant

Hard coding a key value, password, passphrase, or any other sensitive information for that matter, is strongly discouraged. There are many reasons why this is a bad idea. If you are going to hard code your key values, stop reading right now. There is no point.

- ▶ In a program variable

Within your application, key values should always be handled in variables. Ideally, the key value in the variable should be encrypted. That way, if the program is viewed while running (for example, it takes a dump), there is less chance that the key value will be exposed.

- ▶ Offline

- On non-digital media

One way of protecting a key value is to not store it digitally at all. Instead, the user enters the key whenever it is needed. Of course, the drawback in this scenario is the interactive requirement.

Also, it does not mean that there are no security issues to address. Is the key written down, and is that stored securely? Whether it resides on paper or in someone's head, is there a backup? How will the key be entered? Is the setting secure? Can anyone view the typing? Is the application set up so that the value does not appear in the window or in the joblog?

One common method of entering a key is to use a passphrase and then apply some cryptographic functions to create a key value, such as with the RSA Data Security, Inc., Public-Key Cryptography Standard (PKCS) #5 algorithm. As discussed above, you need to ensure that enough characters are entered to obtain sufficient entropy.

- On digital media

Keys used to encrypt data are often stored with the data itself so that they cannot be misplaced. However, the key should not be stored in the clear. The key must be encrypted with at least the same level of security as the data.

- ▶ In an i5/OS object

We have two recommendations for securing keys stored in an i5/OS object, such as in a file or data area:

- Encrypt the key values.

If your object is saved to media, you do not want those keys exposed. Secondly, it provides another layer of defense while the object is on your system. If an unauthorized user gets access to the object, this person will not be able to see the key values.

Of course, the question now becomes how to store and manage the key that encrypts the key values. This issue will be discussed in more detail in 3.5.1, “Key hierarchy” on page 30.

- Lock down access to the object.

Even though the key values in your object may be encrypted, a user may have access to the decryption function and the key-encrypting key.

- ▶ In an i5/OS-provided keystore

This book discusses two i5/OS-provided keystores:

- CCA keystore for use with the cryptographic coprocessors via the CCA APIs

- Cryptographic Services keystore for use with the Cryptographic Services set of APIs

Both keystores comprise a set of database files. All key values within the keystore files are stored encrypted with a master key.

It is important to secure i5/OS objects that contain keys, including the provided keystores. Refer to 5.4, “Establishing a secure keystore environment” on page 65, for a detailed discussion.

## 3.5 Key separation

An important aspect of key management is good key separation. Keys should be separated by the data they act on (keys or data), by key use (encrypting, decrypting, message authenticating, signing, and so on), by authorities, and by key management responsibilities.

### 3.5.1 Key hierarchy

Besides encrypting data, keys are used to encrypt other keys. All keys should be stored encrypted. Keys should always be transmitted encrypted, and applications should handle encrypted keys as much as possible.

Figure 3-1 depicts a hierarchical key system, a commonly used key management technique. A key hierarchy is a popular mechanism for protecting keys primarily because it reduces the number of secrets that are in the clear.

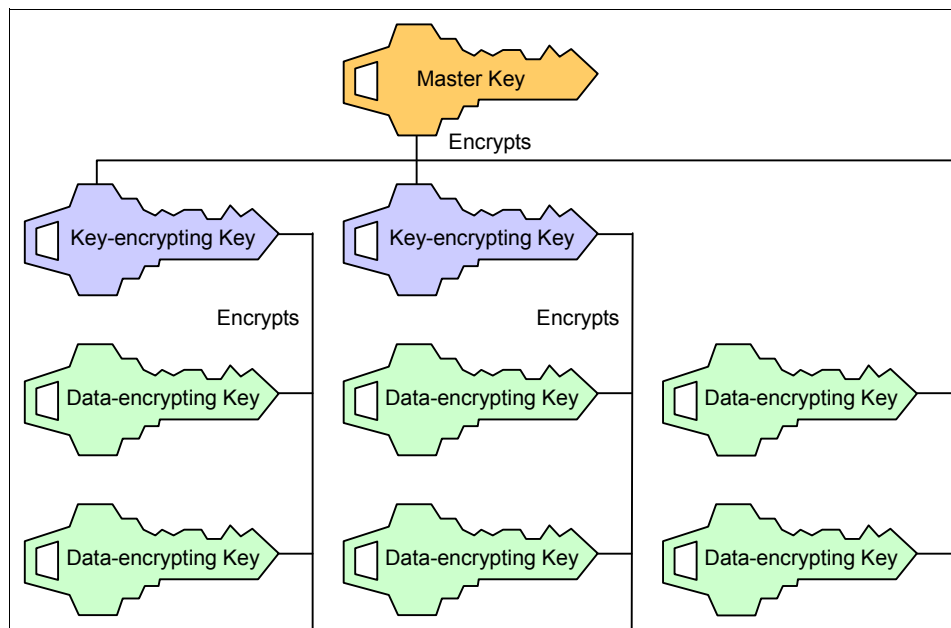


Figure 3-1 Key hierarchy

At the top of the hierarchy is a *master key* (or keys). The master key is the only clear key value and must be stored securely. CCA stores master keys in tamper-responding hardware. Cryptographic Services stores master keys in a protected area of the Licensed Internal Code (LIC). If you are managing your own keystore, you need to decide how to securely store a master key. One method is to enter it into the system whenever a key needs to be decrypted, as discussed in 3.4, “Storing keys” on page 29. Or you could use a Cryptographic Services key to encrypt keys in your keystore.

Use *key-encrypting keys (KEKs)* to encrypt other keys. KEKs are usually stored encrypted under a master key.

*Data keys* are used directly on user data (for example, to encrypt or sign data). You can encrypt a data key under a KEK or under a master key. When data keys are stored in keystore, they are encrypted under a master key. When a data key is sent to another system or stored with the data that it encrypts, it is usually encrypted under a KEK.

### 3.5.2 Key use

Keys for different purposes should be separated cryptographically. For example, if an application only needs to encrypt, the key should be limited to encryption only. This is accomplished with the use of *control vectors*.

A control vector is a non-secret value that is exclusive-ORed with a master key or a KEK before using that master key or KEK to encrypt or decrypt another key, as shown in Figure 3-2.

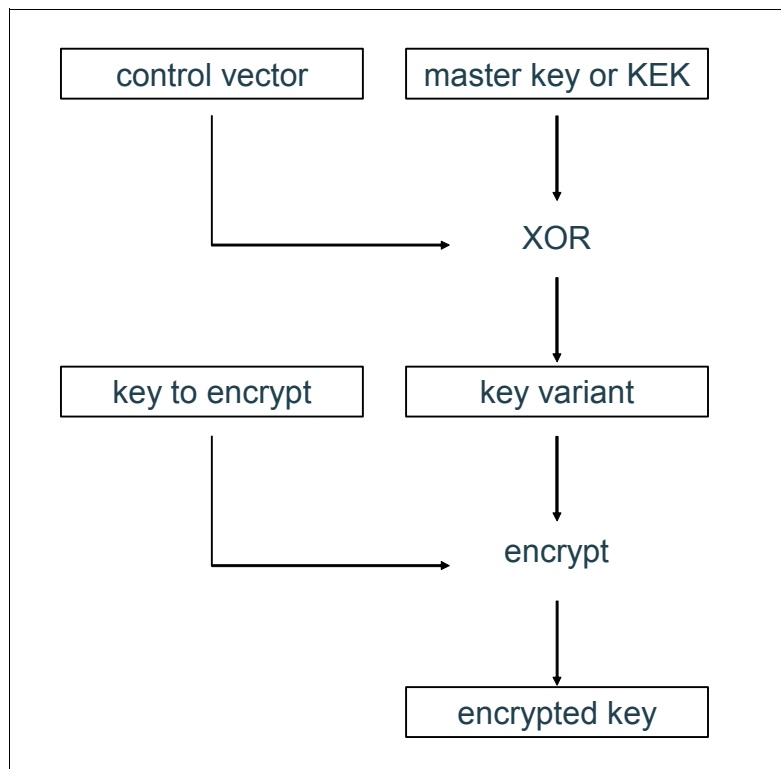


Figure 3-2 Using a control vector

The control vector carries information about how the key can be used. Because the control vector is used in the encryption of the key, it permanently binds that information to the key. In this way the control vector restricts the use of the key to certain operations. The control vector must remain with the key and is supplied on the cryptographic operation.

A control vector prevents unauthorized use of the key in two ways:

- ▶ If an attempt is made to use a key on an unauthorized operation, the operation will detect that the supplied control vector disallows the request and will terminate the operation.
- ▶ If the control vector is altered so that the operation allows the request, the key will decrypt incorrectly, causing bad results.

CCA supports a very comprehensive control vector, encompassing over 50 types of key usage. A CCA control vector can be applied to a key encrypted with a master key or encrypted with a KEK.

Cryptographic Services supports a much simpler control vector. A Cryptographic Services control vector allows you to prevent a key from encrypting, decrypting, message authenticating, signing, or any combination of these four functions. A Cryptographic Services control vector can only be applied to a key encrypted with a master key.

### 3.5.3 Keystore authorization

A further means of separating keys is to place them in different keystore objects and limit the authorizations placed on those objects. This is important because a user profile that has authority to a keystore in which it only uses a single key, will have access to all the keys. Refer to 5.4, “Establishing a secure keystore environment” on page 65, for a detailed discussion on securing keystores.

### 3.5.4 Key management responsibilities

No single person should have responsibility for all key management operations. As part of your key management planning, decide who will be responsible for what operations. Consider the following:

- ▶ Who will set your master keys?

As discussed above, no one person should know the value of, or be able to reproduce, a master key. Master key parts should be assigned to separate individuals.

- ▶ Who is responsible for translating keys when a master key is changed?

The system does not keep track of your keystore files. You must know what files need to be translated when a master key is changed. It is not a good idea to give one person authority to all keystore files.

- ▶ Who will change keys, as outlined in 3.7, “Changing keys” on page 33, and do any necessary translations?

- ▶ Who is responsible for checking for unauthorized changes?

Audit records are cut for master key operations so that you can track when master keys are altered. Also, someone could check the key verification value (KVV) of master keys to determine whether master keys have changed. For more information about master key KVV, refer to “Clear Master Key” on page 50.

- ▶ Who is responsible for backing up master keys and keystore files?

## 3.6 Backing up keys

It is essential to keep a current backup of all keys.

How you back up your master keys is dependent on where your master keys are stored. Refer to Chapter 5, “Managing keys on System i” on page 47, for specifics on backing up your master keys.

CCA and Cryptographic Services keystore files can be saved with the SAVOBJ or SAVLIB commands.

Anytime a new key is added to a keystore file, you should make a backup of the file.

Anytime you change a master key, it should be backed up. In addition, keys encrypted under that master key must be translated and backed up.

Do not forget about keys stored outside of keystore. These should be backed up as well. Generally, these should not be encrypted under a master key. If you store a key outside of keystore, it is best to encrypt it under a KEK. If you encrypt it under a master key and that master key is changed, you must remember to translate and back up the key.

## 3.7 Changing keys

How often you change cryptographic keys is another important consideration in your key management strategy. A number of factors affect the *cryptoperiod*, or life-span, of a key:

- ▶ Strength of the cryptographic algorithm

For example, if you currently have data encrypted under a DES (56-bit) key, you should consider translating the data under another algorithm that supports a larger key size.

- ▶ Operating environment

If you do not have a secure environment for creating new keys and performing translations, it may be better to leave the key as is.

- ▶ Frequency or amount of data acted upon

If a key is used infrequently on a small amount of data, it need not be changed as often as a key that is used frequently or on large amounts of data.

- ▶ Sensitivity of the data

What are the consequences of exposure? The greater the consequences, the shorter you should make the cryptoperiod.

- ▶ Cost of translation

Generally, the cryptoperiod is longer for keys that encrypt stored data than for keys that encrypt communications exchanges. In some cases, replacing a key and forcing the translation of a large amount of data is too costly.

- ▶ Cryptographic operation

How is the key used? A key used for encryption-only can be changed frequently. On the other hand, a key used for decryption of stored data may need a much longer cryptoperiod, depending on the life span of the encrypted data.

- ▶ Possible key compromise

If there is a possibility that a key is compromised, the key should be changed immediately. For example, someone with knowledge of a master key part who leaves the company is an exposure to the master key.

In general, a shorter cryptoperiod increases the security of the encrypted data.

The following is a simplified version of the general recommendations for cryptoperiods by NIST.<sup>2</sup> These recommendations should be evaluated in light of the risk factors listed above:

- ▶ Master keys

Master keys should be changed at least once a year.

<sup>2</sup> See *SP 800-57 Part 1, Recommendation for Key Management - Part 1: General (Revised)*, found at [http://csrc.nist.gov/CryptoToolkit/kms/SP800-57Part1\\_3-8-07.pdf](http://csrc.nist.gov/CryptoToolkit/kms/SP800-57Part1_3-8-07.pdf) for a more detailed discussion.

- ▶ KEKs
 

For encryption of a large number of keys over a short period of time, ideally, the cryptoperiod should be between one day and one week. For encryption of a smaller number of keys, the cryptoperiod may be as long as a month.

For decryption, the cryptoperiod can be significantly longer.
- ▶ Symmetric data-encryption keys
 

For encryption of a large amount of data over a short period of time, ideally, the cryptoperiod should be between one day and one week. For encryption of smaller amounts of data, the cryptoperiod may be as long as a month.

For decryption, especially of stored data, the cryptoperiod can be significantly longer.
- ▶ All other keys
 

For all other keys, the cryptoperiod can be from one to two years.

## 3.8 Key distribution

There are several alternatives for distributing a key (or password/passphrase) securely:

- ▶ Deliver the key via a secure physical channel.
 

For example, exchange a key face to face, or use a bonded courier.
- ▶ Send the key over a secure session, such as SSL with client authentication.
 

SSL with client authentication authenticates both parties. In addition, when the key is sent, it will be encrypted automatically.
- ▶ Encrypt the key yourself to send over an insecure channel.
 

RSA, a PKA algorithm, is often used to encrypt key values. The public key can be sent in the clear and the recipient can then use it to encrypt a symmetric key. The encrypted symmetric key can only be decrypted by the party who possesses the private key.
- ▶ Use Diffie-Hellman to jointly establish a key.
 

Diffie-Hellman is also a PKA algorithm, but it is not used to encrypt a key like RSA. It is used to jointly establish a key. Basically, someone generates a set of PKA parameters. These parameters are not secret and are distributed in the clear. Each party uses the parameters to generate a private/public key pair. Both parties calculate the shared secret key using their own private key and the other party's public key.
- ▶ Move the i5/OS object containing the encrypted key.
 

If you are sending your key to another i5/OS, and your key is stored (presumably encrypted) in an i5/OS object, such as a Cryptographic Services keystore file, then you can simply move your object to the other system via i5/OS's save/restore functions. Of course, whatever key (KEK or master key) your key is encrypted under, that must be on the target system as well.

Which key distribution method should you use? It depends on your particular situation. Be aware that the third and fourth options by themselves do not provide any sort of authentication. Getting authentication correct when developing a key exchange protocol is tricky and extremely important. Done right, it can be more secure than the other alternatives. Designing a secure key exchange protocol is beyond the scope of this book.

Using a secure authenticating protocol, like SSL with client authentication, is the easiest method for securely exchanging a key value because the protocol authenticates the



communicating parties for you. Be aware, however, that just because you use SSL, it does not mean that your key is secure. Consider the following:

- ▶ SSL security relies on the authentication of certificates. Many SSL clients (in particular, most Web browsers) are too permissive, having too many trusted root certificates and accepting almost any certificate they receive.
- ▶ Even though SSL encrypts all data sent over the session, it is still a good idea for you to encrypt the key being exchanged. That way you can avoid exposing the clear key value as it travels through the system. For example, if the key that you wish to distribute resides in a keystore file, perform an export operation to translate the key under a KEK, such as an RSA public key, and then send it via SSL.

Another option is to use Diffie-Hellman over SSL.

- ▶ You should verify the key value before it is used. Just because the session is authenticated and the key is encrypted does not mean that it cannot be altered in transit.

## 3.9 Key destruction

All unnecessary key material should be destroyed when no longer needed. This applies to both permanent and temporary storage of keys, as discussed in 3.4, “Storing keys” on page 29. For example, if your application holds a key value in a variable, that variable should be cleared as soon as you are done with it. Or a key that has been distributed to another system and is no longer needed on the source system should be deleted or changed.

You should track carefully what keys are used for what data so that you do not delete a key prematurely. If unsure, do not delete the key. If you lose a key and have no backup, all data and keys encrypted under that key will be lost.





## Part 2

# Planning for data encryption

This part provides critical information for planning a data encryption project on i5/OS.





## Analyzing needs and defining scope

Encryption is not a task that should be taken on lightly, and deciding what and how to encrypt your data, not to mention why, are items that need to be carefully considered.

Not unlike any development project, the evaluation of encryption technology needs to progress through defining a need and evaluating the impact and associated costs to complete.

This chapter first introduces some of the common legislation pertaining to data security, and then outlines some topics for evaluating how encryption can be deployed in your organization.

## 4.1 Needs analysis

The fact that you are reading this book indicates you already are aware, or at least suspect, that you have a requirement for encrypting data. Before you embark on your project, the requirements must be fully understood. Not only will they affect what and how you encrypt, but many other security decisions need to be made as well. For example, do you need to protect data from users who have access to the system? If you use the Cryptographic Services APIs, but everyone has All Object (\*ALLOBJ) special authority, you gain no protection while that data resides on the system. Someone with \*ALLOBJ authority has access to the APIs and keys needed to decrypt the data. However, if you only care about protecting data outside the perimeter of your system security, for example, saving the encrypted data to tape, it will be protected by those who do not have access to the system.

As discussed in 1.1.2, “What drives the requirement” on page 4, there are three main requirements that lead to encryption: government legislation, industry rules, and business requirements.

### **Regulations and standards**

This section briefly outlines some of the most notable compliance requirements—both government and industry—pushing organizations to analyze their security infrastructure.

For a more in-depth review of regulations and standards, refer to the Redbooks publication *IBM System i Security Guide for IBM i5/OS Version 5 Release 4*, SG24-6668.

### ***Sarbanes-Oxley (SOX) Section 404***

The Sarbanes-Oxley act (also known as SOX and Sarbox) was signed into effect July 30, 2002, after a series of corporate financial scandals. It places stringent financial reporting requirements on publicly traded companies doing business within the U.S. Section 404 concerns assessment of internal controls.

Encrypting and authenticating data can fulfill some of the accountability requirements of this law.

For more information about SOX, see the Securities & Exchange Commission Web site:

<http://www.sec.gov/spotlight/sarbanes-oxley.htm>

Also see the Sarbanes-Oxley Web site:

<http://www.sarbanes-oxley.com>

### ***Gramm-Leach-Bliley Act (GLBA)***

The GLB Act was passed in 1999 to repeal a previous U.S. Congressional Act that prevented banks from offering investment and insurance services. This opened the way for mergers of corporations operating in these sectors, most notably the Travellers Insurance buyout of Citibank to form CitiGroup.

The Act also defines several rules that control how institutions maintain the private information of individuals, including disclosure of how that information is shared, the protection of any stored information, and the prevention of data access by false pretenses.

Section 6801 (Protection of non-public personal information) of the Act (also known as the Safeguards Rule) requires that financial institutions develop and maintain a written security policy regarding how they are protecting clients' private information. According to the U.S. Federal Trade Commission, this policy should be designed:

- ▶ To insure the security and confidentiality of customer records and information
- ▶ To protect against any anticipated threats or hazards to the security or integrity of such records
- ▶ To protect against unauthorized access to or use of such records or information that could result in substantial harm or inconvenience to any customer

For more information, refer to the U.S. Federal Trade Commission (FTC) Web site:

<http://www.ftc.gov/privacy/privacyinitiatives/blbact.html>

### **Healthcare Insurance Portability and Accountability Act (HIPAA)**

Enacted by the U.S. Congress in 1996, HIPAA is designed to protect workers' right to health insurance when they change or lose their job, as well as to establish national standards related to the security and privacy of health-related information.

Title II of the Act covers the requirements regarding security and privacy. The Security Rule has multiple elements that pertain to computer systems including administrative safeguards, physical safeguards, and technical safeguards. The technical safeguards dictate that the transmission of Protected Health Information (PHI) across open networks must be encrypted, while encryption of data on closed systems is currently optional.

For more information about HIPAA, refer to the U.S. Department of Health & Human Services Web site:

<http://www.hhs.gov/ocr/hipaa/>

### **Payment Card Industry (PCI) Security Standards**

The Payment Card Industry was established by the four major credit card corporations (American Express, VISA, MasterCard, and Discover) in response to breaches of personal information leading to increased incidences of identity theft.

The purpose of the standards is to provide consumers with reassurance that their personal details are safe when provided to a participating company or merchant. This includes the requirement that credit card information be encrypted and the ban on retention of certain card-related details (for example, CVV security code).

Table 4-1 is from the PCI data security standards (DSS) guidelines v.1.1.1, published in September 2006. The PCI DSS standard only applies if a primary account number (PAN) is stored, processed, or transmitted.

*Table 4-1 PCI DSS requirements*

	<b>Data element</b>	<b>Storage permitted</b>	<b>Protection required</b>	<b>PCI DSS req. 3.4</b>
Cardholder data	Primary account number (PAN)	Yes	Yes	Yes
	Cardholder name*	Yes	Yes*	No
	Service code*	Yes	Yes*	No
	Expiration date*	Yes	Yes*	No

	Data element	Storage permitted	Protection required	PCI DSS req. 3.4
Sensitive authentication data**	Full magnetic stripe	No	N/A	N/A
	CVC2/CVV2/CID	No	N/A	N/A
	PIN/PIN block	No	N/A	N/A

\* These data elements must be protected if stored in conjunction with the PAN. This protection must be consistent with PCI DSS requirements for general protection of the cardholder environment. Additionally, other legislation may require protection of this data.

\*\* Sensitive authentication data must not be stored subsequent to authorization (even if encrypted).

Compliance requirements vary between card networks, as each brand has its own format, as follows:

- ▶ American Express - Data Security Operating Policy (DSOP)
- ▶ MasterCard - MasterCard Site Data Protection (SDP)
- ▶ VISA - Cardholder Information Security Protection (CISP)
- ▶ Discover - Discover Information Security and Compliance (DISC)

Each brand also defines the requirements for four levels of merchants, based primarily on the number of annual transactions and the compromise history. The penalty for non-compliance may include increased processing fees, loss of ability to accept payments, and, in the case of serious data breaches, fines of up to \$500,000. In December 2006, VISA announced that they were offering \$20 million in incentives to those merchants demonstrating compliance in order to accelerate progress.

For more information about PCI Security, visit the PCI Security Standards Council Web site:

<http://pcisecuritystandards.org>

### ***Notice of Breach (California Senate Bill 1386)***

Passed in 2003 after a data breach of California's state payroll database, SB-1386 requires that any organization (business or government) that maintains a database of personal information (as defined by the Bill) on a customer or employee that resides in California be required to notify individuals in a timely manner if that data is compromised.

In essence, personal information is defined as the storage of first name (or initial) and last name in combination with uniquely identifying information such as Social Security number or drivers licence number.

The bill had far-reaching effects as other U.S. states reacted upon the realization that unauthorized disclosure of private information about their own state residents could legally go unreported.

Significant to the discussion in this Redbooks publication, the SB-1386 disclosure requirement is limited to compromised data that is not encrypted with at least 128-bit encryption. For more information, refer to the California State Senate Web site:

[http://www.info.sen.ca.gov/cgi-bin/postquery?bill\\_number=sb\\_1386&sess=PREV&house=B&site=sen](http://www.info.sen.ca.gov/cgi-bin/postquery?bill_number=sb_1386&sess=PREV&house=B&site=sen)



### ***North American Electric Reliability Council Cyber Security Standards***

NERC acts as an intermediary to coordinate cooperation between the electric industry and the U.S. federal government. Its main function is to develop protection for the U.S. electric system from both physical and cyber attacks by working with the various agencies including the U.S. Department of Energy and U.S. Department of Homeland Security.

NERC adopted its current Cyber Security standards in 2006, which are designed to protect critical data exchanges. The standard is one of several designed to establish best practices. For more information, refer to the NERC Web site:

<http://www.nerc.com>

### ***UK Data Protection Act for UK and European companies***

The Data Protection Act of 1984 (and subsequently 1998) pertains to storage and use of personal identification information of any living UK resident.

Although there are some exceptions, data may not be used outside of the purpose for which it was collected, and may not be disclosed without the consent of the owner of the information. The information in the data must be current, accurate, and retained only as long as the data is required. The act also enables an individual to pay a nominal fee to review the data that is recorded about them, and request corrections to inaccuracies.

As with most legislative requirements, any personal data must be stored securely, although the requirement to encrypt data is not spelled out.

Failure to abide by the provisions in the act can result in financial penalties as well as the requirement to destroy the information.

For the full text of the 1998 revision of the DPA, refer to the Office of Public Sector Information Web site:

<http://opsi.gov.uk/acts1998/19980029.htm>

## **4.2 Defining the scope**

If your interest in encryption is compliance-driven, then start scoping your encryption project requirements specifically around the data impacted by the compliance regulations.

If not spelled out by compliance, evaluate your data based on whether it would cause harm or embarrassment to the organization if it were published in a forum, such as a public Internet site or trade magazine.

You should keep in mind that unauthorized access to even unimportant corporate data would likely bring into question the security of the entire database, so good general security practices are mandatory.

### **4.2.1 What data to protect**

Your analysis includes assessing what data should be encrypted.

First, the feasibility of encrypting *all* data in the database is going to depend on several factors:

- ▶ Performance

Performing any cryptographic function is going to add an overhead to your processes. Regardless of whether you make application modifications or use database file triggers, the more information that you have to encrypt (and subsequently decrypt), the more overhead you add. Focussing *only* on the data that is deemed as truly confidential minimizes the impact.

- ▶ Scope

Your application is likely going to require changes to perform the cryptographic functions. Although it is possible to do this via triggers, restricting the files and fields to be processed to the critical ones reduces the number of modifications or triggers that need to be written. In other words, simply reading/writing encrypted data (that is, not decrypting it) does not add more cycles than are required for plaintext information. But the more fields you encrypt, the more likely the requirement to decrypt it, and therefore the requirement to write the code to perform the decryption.

- ▶ Security

Encrypting low-risk information can potentially reduce the added security afforded by encryption. For example, if an address in a customer file is encrypted, but that information is readily available in plaintext somewhere else in the database, then you have provided a link that could lead to your entire encryption being compromised. This would also leave your high-risk information vulnerable.

In addition, much of the data stored in databases often has no intrinsic value outside of the organization that is hosting it and, therefore, has no requirement to be protected beyond the normal security mechanisms afforded by i5/OS. An example might be the customer shipping preferences used during an order entry process.

In fact, some might argue that if the security infrastructure is established correctly, then encryption is not adding much protection anyway (conversely, if the security infrastructure is *not* established correctly, then you are *definitely* adding very little benefit with encryption).

While this observation may be especially true in the case of the System i, you may still be required to demonstrate compliance to a general standard imposed by legislation or governing body, or just to instill consumer confidence.

## 4.2.2 Define your requirements

First, understand that defining exact encryption requirements is not a task that is going to be accomplished by a single person. While security projects are often placed on the shoulders of IT staff, interpretation of the legal directives, along with assessment of the ramifications of non-compliance, is not appropriate for an IT staff member.

You are going to be far more successful if you form a committee or task force that oversees the development of the requirements.

The types of staff to include in the evaluation and planning stages might be:

- ▶ Application programmer
- ▶ Systems programmer
- ▶ Security officer
- ▶ IT Management
- ▶ Auditor
- ▶ Corporate attorney

Your objective is to bring together the people who know what is required for compliance, with those who know which of those requirements pertain to the organization. A sound understanding of the database and the data stored within it are critical.

Assessing what data is a good candidate for encrypting is based on these established requirements.

### 4.2.3 Evaluate the impact of change

Once the requirements are defined, you must determine the impact on the existing data and applications.

If you use a third-party application provider, you will want to check and see whether it has a release of its application that addresses compliancy requirements. If this is the case, much of the discussion will be simplified, although you will still want to understand how the vendor approaches their solution, including key management techniques, and whether they understand and apply a strong base i5/OS security infrastructure to build upon.

Much of the evaluation is going to be similar to that of any application modification and should address questions such as:

- ▶ Does your organization own the source code for the applications that access the data?  
If the source code is not accessible, then it is going to be difficult to perform database encryption. Using SQL views with INSTEAD OF triggers might be one solution.
- ▶ Do you have a secure i5/OS environment (that is, can you protect the encryption keys)?  
If your i5/OS environment is not secure then how are you going to prevent unauthorized access to the keys and decryption routines?
- ▶ What key management techniques are going to be utilized?  
You have choices about what types of keys to use, as well as in which mechanism to store your keys: an i5/OS keystore file, a self-managed keystore, a hardware-based key management device, or even SQL-based passwords.
- ▶ Are there existing staff with the skills (and time) necessary to perform modifications?  
Are you going to have to hire outside resources, and how accessible are those skills in the market?
- ▶ What interfaces are currently used to access the data (native, SQL, ODBC, and so on)?  
Awareness of how your data is used by *all* applications is necessary to ensure that no mission-critical access method is overlooked.
- ▶ What transports are used to move the data off the server (tape media, electronic, and so on)?  
If you store the data encrypted in the database and then transmit plaintext across insecure transport mechanisms, then you are exposed.

- ▶ What approach is going to be used to modify the database?  
If changes are necessary, you have to decide how the structure will be designed to support the encrypted information.
- ▶ What is the impact of converting the existing data and validating the results?  
Ensuring that the initial conversion is correct is critical, as is assessing how the conversion window will impact your application availability.
- ▶ Are there quality assurance (QA) or test systems that are going to be affected?  
Do not move encrypted data to test systems and then store in plaintext.
- ▶ Where is the data utilized within the application?
- ▶ Are there reports that include private data?  
Securing reports and eliminating unnecessary private information can prevent a simple spooled file review from revealing confidential information.
- ▶ How will this impact system recovery and high availability functions?  
Make sure that HA rollover and disaster recovery tests include additional steps for key recovery and data access.
- ▶ Are there mechanisms that are going to be incompatible with encrypted data?
- ▶ Do you rely on any tools or applications that are not compatible with encrypted data in the files?

One of the biggest reasons that data leaks is that many organizations do not fully understand where their information is stored, and how it is accessed and by whom. Once the impact of the changes are established, the estimated cost of addressing each of them can be ascertained.

#### 4.2.4 Return on investment

Compare the cost of making the necessary changes and infrastructure investments against the estimated cost of *not* encrypting. This cost may be manifested in non-compliance fines or, more likely, in the financial and political cost to the business incurred from a data breach (or possibly both).

Depending on the type of data that is breached, these costs may include:

- ▶ Notification costs for victims
- ▶ Identity theft protection fees
- ▶ Loss of confidence by consumers and business partners
- ▶ Corporate embarrassment
- ▶ Fines for non-compliance
- ▶ Restriction of business function
- ▶ Lawsuits filed by data owners
- ▶ Complete business failure

Take any combination of these, and you will often find that the cost of being reactive far exceeds the cost of being proactive.

As mentioned previously, much of the above is a statement of the importance of data security in general, as it is a statement about encryption. However, encryption provides an additional layer of protection that reduces the impact of a breach, in the event that one does happen.



## Managing keys on System i

Assuming that you are using appropriate algorithms and key sizes, an attack by breaking an algorithm or performing an exhaustive key search is unlikely. More likely, an attacker will look for an improperly protected key. This chapter describes how to protect your keys with three kinds of key management: Cryptographic Services, CCA, and roll-your-own.

## 5.1 Cryptographic services

This section discusses the implementation of cryptographic services on i5/OS.

### 5.1.1 Master keys

i5/OS Cryptographic Services supports eight master keys. These master keys are used to encrypt other keys (KEKs and data keys), but not data. The master keys are stored in the i5/OS Licensed Internal Code (LIC) in an area that cannot be displayed or even viewed via dump service tools. You can access the master keys only with the Cryptographic Services APIs. To reference a master key on an API, you simply use a number, 1–8.

#### Master key versions

A Cryptographic Services master key is a 256-bit (32-byte) AES key. Each master key has three *versions*:

- ▶ New

The *new* master key version contains the value of the master key while it is being loaded.

- ▶ Current

The *current* master key version contains the active master key value. The current version is the version that is normally used on a cryptographic operation.

- ▶ Old

The *old* master key version contains the previous current master key version. It is used to prevent the loss of data and keys when the master key is changed.

Each version also contains a *key verification value (KVV)*, a 20-byte hash of the key value.

Figure 5-1 depicts the structure of a master key.

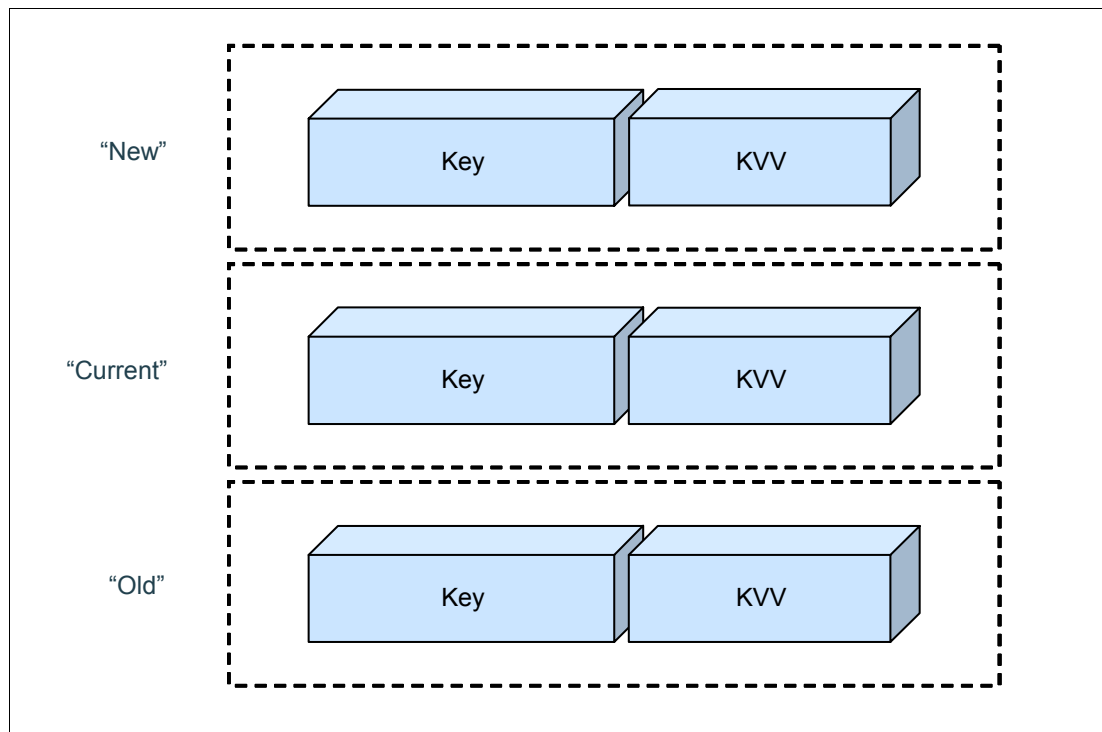


Figure 5-1 Structure of a master key

## Master key APIs

Cryptographic Services supports four APIs for managing master keys:

- ▶ Load Master Key Part

The Load Master Key Part API takes a passphrase as input. It is hashed and then loaded into the new version. You can load as many passphrases as desired. Each passphrase is XORed into the new version of the master key. To ensure that no single individual has the ability to reproduce a master key, you should assign passphrases to several people.

- ▶ Set Master Key

The Set Master Key API activates the new master key value, which consists of the passphrases previously loaded. The following steps are performed:

- a. The current version master key value and KVV are moved to the old version, wiping out what was in the old version.
- b. The new version master key value is finalized. Then it and its KVV are moved to the current version.
- c. The new version is erased.

The Set Master Key API returns the master key KVV. You should retain this value so that at a later date you can determine whether the master key has been changed.

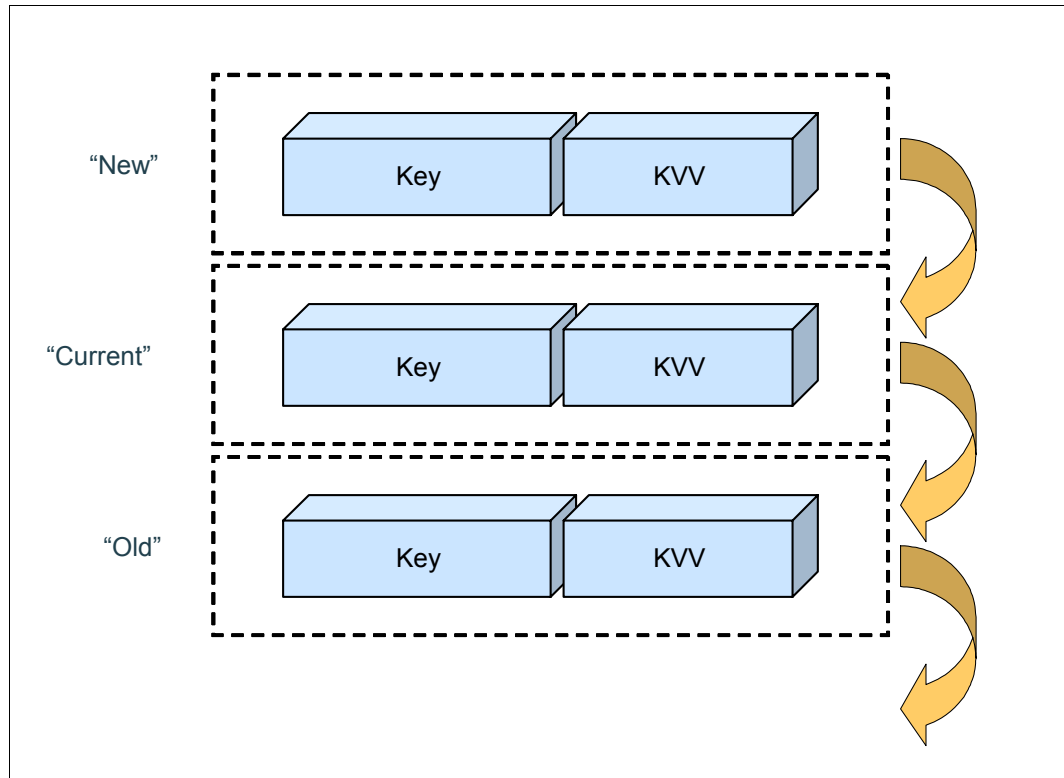


Figure 5-2 Setting a master key

► Clear Master Key

The Clear Master Key API clears the specified master key version. Before clearing an old master key version, care should be taken to ensure that no keys are still encrypted under it.

► Test Master Key

The Test Master Key API returns the key verification value (KVV) for the specified master key. You can compare this to the KVV returned on the Set Master Key API to determine whether the master key value has changed.

### Master key KVV

A master key KVV is a 20-byte hash of the master key value. A KVV is kept with each version of a master key.

Whenever a key is encrypted under a master key, the KVV for the current version of the master key is returned. When a key is stored in a keystore file, the KVV of the master key is stored in the key record along with the key value. When a key encrypted under a master key is stored outside keystore, it is best to store the KVV with it.

When a key encrypted under a master key is used on an API and the master key KVV is supplied, cryptographic services will check the supplied KVV against the master key versions' KVV. If the supplied KVV matches the current version KVV, the operation will proceed normally. If the supplied KVV matches the old version KVV, the operation will proceed but return a diagnostic to the API informing the user that the key needs translation. If the supplied KVV matches neither, the operation will end with an error.

If no KVV is supplied on an API for a key encrypted under a master key, the current version will be used to decrypt the key.



**Note:** A key hierarchy is a popular key management technique primarily because it reduces the number of secrets to protect. Be discriminating about how many master keys you actually need to use and must therefore manage.

## 5.1.2 Keystore files

You can securely store KEKs and data keys in a Cryptographic Services keystore file. A keystore file is a database file that you create with the Create Key Store API. You specify the file name and library, the public authority, and the master key that encrypts all the key values stored in the keystore file.

To store a key in a keystore file, use the Write Key Record API. Parameters let you specify the type and size of key; whether the input key value is a binary string, a BER-encoded string, or a Privacy Enhanced Mail (PEM) certificate; whether the input key value is encrypted and, if so, the KEK information; and a label for referencing the key. Or you can use the Generate Key Record API to generate and store a random key value in a keystore file, again specifying key type and size and a label for the key record.

Any type of key supported by cryptographic services can be stored in a keystore file. Keys are stored in one of two formats:

- ▶ Secret key format

The secret key format includes all symmetric and HMAC keys.

- ▶ PKA key format

The PKA key format stores an RSA public/private key pair, or just an RSA public key.

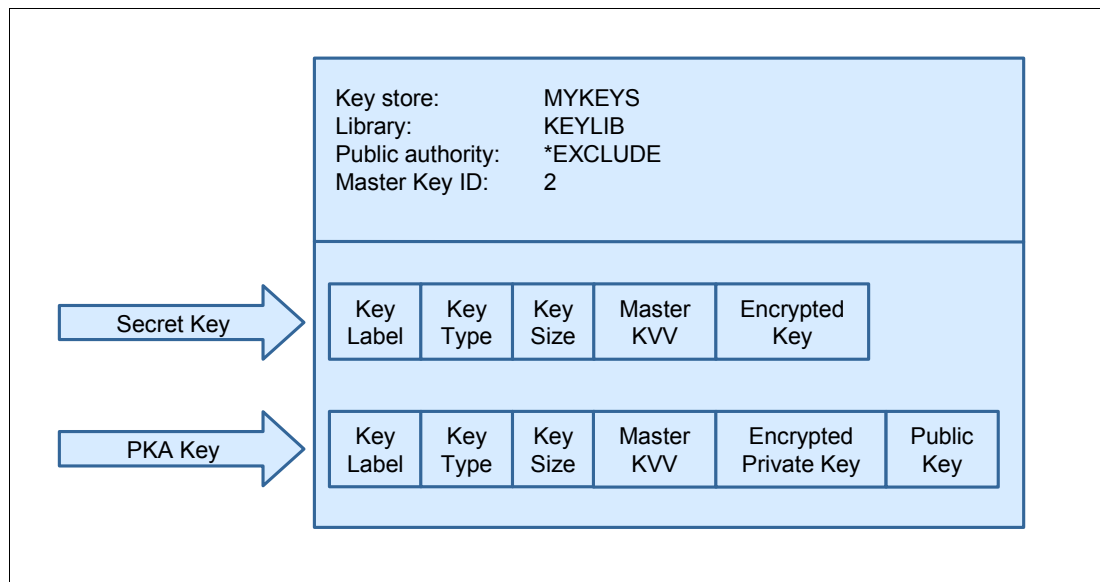


Figure 5-3 Cryptographic Services keystore

When creating keystore files, give careful consideration to authorities. Even though key values in a keystore file are encrypted, anyone with access to the keystore file and the appropriate API (for example, the Decrypt Data API) could hack the data. Note, however, that a keystore file moved to another system is useless if the master key under which it was encrypted has not been set identically. For more help on securing your keystore files, refer to 5.4, “Establishing a secure keystore environment” on page 65.

If a master key for a keystore file is changed, the keys in that file must be translated. The Translate Key Store (OPM, QC3TRNKS; ILE, Qc3TranslateKeyStore) API can be used to translate keystore keys to another master key, or if the same master key is specified, to the current version of the master key.

### 5.1.3 Changing a master key

Change a master key by loading key parts and setting the master key as described in “Master key APIs” on page 49.

Whenever a master key is changed, all keys encrypted under that master key require translation. For keystore files, use the Translate Key Store (OPM, QC3TRNKS; ILE, Qc3TranslateKeyStore) API. For keys stored outside a keystore file, use the Export Key then Import Key APIs.

### 5.1.4 Master key variants

You can limit the use of a master key encrypted key by using a master key *variant*, or control vector. For example, if your application program uses a key from keystore to encrypt sensitive data, you can use variants to ensure that the key cannot be used to decrypt the data.

A master key variant is a value that is XORed into the master key value before encrypting a key. Refer to Figure 3-2 on page 31.

On APIs that encrypt a key with a master key (such as the Generate Key Record API), specify a master key variant via the Disallowed Function parameter or field. Functions that you can disallow are encrypt, decrypt, Message Authentication Code (MAC and Hash MAC), sign, or any combination of these.

When a key that has been encrypted under a master key variant is used on an operation (for example, decrypting data), the variant must be supplied to properly decrypt the key. The master key variant for a keystore file key is stored in the key record and picked up automatically when that key is specified on an API. For keys outside of keystore, you must manage and supply the master key variant yourself. If no variant is supplied, the default is to allow all functions.

If the supplied variant indicates that the operation is disallowed, an error is returned. If the variant is altered to force the operation, the results will be bad. For example, if you use an altered variant on the Decrypt Data API, the decryption proceeds normally, but the result is not cleartext data.

### 5.1.5 Using keys in an application

In this section we discuss using keys in an application.

## Key formats

Cryptographic Services APIs allow you to specify a key in several ways. The Encrypt Data, Decrypt Data, Calculate MAC, Calculate HMAC, Calculate Signature, and Verify Signature APIs all take a parameter called *Key Description*. Several structures are defined for the Key Description parameter that allow you to specify the following keys:

- ▶ A clear key

Fields describing the key (such as size, type, value) can be specified for any of the above APIs. This is not the preferred method of specifying a key because it exposes the clear key value within the application program. It is better to use one of the other formats.

- ▶ A key in keystore

The file name, library, and label for a key in a Cryptographic Services keystore file can be specified on any of the above-mentioned APIs.

- ▶ A PKCS #5 key

RSA Security's Public Key Cryptography Standard #5 derives a symmetric key from a password, a salt, and an iteration count. The password must be kept secret. The salt value, which need not be secret, is used to produce a large set of key possibilities for each password. Therefore, the salt should be a good random value. The iteration count is used to increase the length of computation. Using a large iteration value makes an exhaustive search for the key prohibitive. You can specify PKCS #5 format on the Encrypt Data, Decrypt Data, Calculate MAC, and Calculate HMAC APIs. For more information about PKCS #5, Password-Based Cryptography Standard, refer to the standard found at the following URL:

<http://www.rsa.com/rsalabs/node.asp?id=2127>

- ▶ A key in a PEM formatted certificate

Privacy Enhanced Mail (PEM) is a standard for secure electronic mail over the Internet. A PEM formatted certificate is a public-key certificate that is base64 encoded and has the text "-----BEGIN CERTIFICATE-----" and "-----END CERTIFICATE-----" appended to the beginning and the end.

**Note:** A public-key certificate is basically a digital ID that can be verified. It contains a serial number, the owner's name, the issuer's name, validity dates, the public key component of the owner's RSA key pair, and a digital signature created by the issuer. Base64 is an encoding scheme in which any arbitrary sequence of bytes is converted into printable ASCII characters.

You can specify a PEM formatted certificate on the Encrypt Data, Decrypt Data, and Verify Signature APIs. For more information about PEM formatted certificates, refer to *RFC 1421, Privacy Enhancement for Internet Electronic Mail*, found at the following URL:

<http://www.ietf.org/rfc/rfc1421.txt>

- ▶ A key from certificate store using:

- A certificate label

A label identifies a public key in a public-key certificate located in i5/OS certificate store. You can specify a certificate label on the Encrypt Data, Decrypt Data, and Verify Signature APIs.

- A distinguished name

A distinguished name (the certificate owner) identifies a public key in a public-key certificate located in an i5/OS certificate store. You can specify a distinguished name on the Encrypt Data, Decrypt Data, and Verify Signature APIs.

- An application identifier

An application identifier identifies the private key associated with a public-key certificate in an i5/OS certificate store. You must use Digital Certificate Manager (DCM) to create an application definition and assign a certificate to it before performing an operation with a private key. You can specify an application identifier on the Encrypt Data, Decrypt Data, and Calculate Signature APIs. For more information about DCM and the i5/OS certificate store, refer to the iSeries Information Center.

- ▶ A key in a key context

A key context is a temporary repository for your key information.

### **Key context**

A key context is a temporary repository for your key information. It is created with the Create Key Context API. The key information can be pulled from a keystore file or a PEM certificate, or specified in PKCS #5 format or in the clear. The API returns an 8-byte token, which can then be specified on other cryptographic APIs to reference your key.

A key context holds key information, such as the key type (for example, MD5-HMAC, AES, RSA), key format (binary or BER-encoded), and key length. Key contexts serve several purposes. Using a key context:

- ▶ Improves the performance for some algorithms such as AES and RC2

Some algorithms require initial key processing prior to performing the cryptographic operation. By using a key context, initial key processing need only be performed once.

- ▶ Allows the application to erase the clear key value from program storage

Erasing the clear key value from program storage will help protect the key.

- ▶ Allows the application to use an encrypted key value with the APIs

Exposure of clear key values within the application program should be reduced as much as possible. Encrypted key values can be used on an API if a key context is created.

A key context is destroyed using the Destroy Key Context API. If not explicitly destroyed, it will be destroyed at the end of the job.

## **5.1.6 Key distribution**

The following information is particular to distributing Cryptographic Services keys. Refer to 3.8, “Key distribution” on page 34, for a more general discussion about the topic.

### **Moving a keystore file**

In general, you should not share master keys with another system. Each system should have unique master keys. However, to move an entire keystore file from one System i to another without exposing clear key values, you need to set up identical master key values on both systems. To avoid exposing your master key values, perform the following steps:

1. Set up a temporary master key on both systems by loading and setting an unused master key with identical passphrases.
2. On the source system and create a duplicate of the keystore file (for example, using the CRTDUPOBJ CL command).
3. Translate the duplicated keystore file to the temporary master key.
4. Move the keystore file to the target system.
5. Delete the translated keystore file from the source system. (You still have the original.)

6. On the target system, translate the keystore file to another master key.
7. Clear the temporary master key on both systems.

### Moving single keys

To move a single key that is encrypted under a master key (in or outside of keystore) to another system, use the Export Key API. The Export Key API translates the key from encryption under the master key to encryption under a KEK. On the target system, you can then use the Write Key Record API to move the migrated key into keystore, or you can use the Import Key API to translate the key value to encryption under a master key. Of course, both systems must agree on the KEK.

**Note:** The Export Key API is shipped with public authority \*EXCLUDE. Be very careful about the access that you give to the Export Key API. Anyone with access to master key encrypted keys and the Export Key API can obtain the clear key values.

## 5.1.7 Generating keys

Cryptographic Services has three APIs for generating key values: Generate Key Record, Generate Symmetric Key, and Generate PKA Key Pair. All three APIs use the system PRNG to generate cryptographically strong pseudorandom key values. Refer to 2.1.4, “Random number generation algorithms” on page 18, for more information.

## 5.1.8 Backing up keys

Keeping a current backup of all your keys is essential. In V5R4, you must back up your master keys by saving their passphrases. Master key passphrases should not be stored on the system in plaintext. Also, do not encrypt them under any of the system's master keys or any key encrypted under a master key. If the master keys are lost (for example, when the LIC is installed) or damaged, you will be unable to recover the passphrases and therefore the master keys. Store the passphrases securely outside the system, such as independently storing them in safes.

Any time a new key is added to a keystore file, you should make a backup of the file. In addition, any time the keystore file is translated, you need to make a new backup of the file.

Do not forget about keys stored outside of keystore. These should be backed up as well. Generally, these should not be encrypted under a master key. If you store a key outside of keystore, it is best to encrypt it under a KEK. If you encrypt it under a master key and that master key is changed, you must remember to translate the key.

## 5.2 CCA key management

The 4758 and 4764 cryptographic coprocessors offer a high degree of security, but with the added cost of complexity. Done properly, using the cryptographic coprocessors with CCA can provide the most secure key management possible. The following sections discuss some of the more salient features of CCA key management. For in-depth reading, refer to *CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors*, available on the following Web site:

<http://www-03.ibm.com/security/cryptocards/library.shtml>

## 5.2.1 Configuring the cryptographic coprocessor

Prior to setting up your key management, you must configure the coprocessor. Do this using the Cryptographic Coprocessor configuration Web-based utility found on the System Tasks page, as follows:

`http://your-server-name:2001`

Or you can write your own application to configure the coprocessor. To set up a cryptographic coprocessor on System i:

1. Create a device description.

The device description is used by the system to direct cryptographic requests to the coprocessor. The device description specifies the specific hardware resource. The device description also specifies a default location for key storage. You can create a device description with or without naming the keystore file for the cryptographic coprocessor.

2. Name the keystore file.

Before you can perform any operation in i5/OS using a keystore file or key stored in a keystore file, you must name the keystore file. You can name a keystore file explicitly by using a program, or you can name it by configuring it in the device description.

3. Set the environment ID (EID) and clock.

The cryptographic coprocessor uses the EID to verify which coprocessor created a key token. It uses the clock for time and date stamping and to control whether a profile can log on.

4. Define and create user roles and profiles.

The cryptographic coprocessors use role-based access control. In a role-based system, you define a set of roles that correspond to the classes of coprocessor users. Next, you enroll each user by defining an associated user profile to map the user to one of the available roles.

Below are some suggestions:

- Set up the coprocessors to enforce a dual-control, split-knowledge policy.
- Carefully consider the coprocessor commands that should be enabled or restricted. Once the node is fully activated, no one person should be able to cause detrimental actions.
- Create a *recovery* profile that can reset master key passphrases. Set this profile to expire a year later than the other profiles and lock it away.
- Triple check all your profiles.
- Delete the default profile after everything is set up.

5. Load a function control vector.

The function control vector tells the cryptographic coprocessor what key length to use to create keys. You cannot perform any cryptographic functions without loading a function control vector.

More detailed information about these steps can be found in the i5/OS Information Center at:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/topic/rzajc/rzajcco4758.htm>

Also refer to the Redbooks publication *iSeries Wired Security, Protecting Data over the Network*, SG24-6168, found at the following Web site:

<http://www.redbooks.ibm.com/redbooks/SG246168.html>

## 5.2.2 Master keys

After you load a function control vector, load and set the master keys. CCA master keys are used to encrypt other keys, not data. They are securely stored within the coprocessor secure hardware module (HSM).

CCA supports two master keys. One is used to protect symmetric keys. The other is used to protect asymmetric keys. The CCA master keys are triple-length (168-bit) triple DES keys.

Each master key is composed of three *master key registers* called new, current, and old. They function like Cryptographic Services master key versions. When loading a master key, the parts are accumulated in the new master key register. After the parts are loaded, the new master key is set by moving the current master key register to the old master key register, and the new master key register to the current master key register, as depicted in Figure 5-2 on page 50.

### Establishing a master key

The easiest and fastest way to load and set master keys is through the Cryptographic Coprocessor configuration Web-based utility found on the System Tasks page at:

`http://server-name:2001`

Or you can write an application program. If writing an application program, use the `Master_Key_Process verb` (or API).

Either way, you have three methods for establishing a master key:

- ▶ Establish a master key using clear information.

You supply individual clear key parts. You can force separation of key part responsibility by setting up the coprocessor so that it requires at least two different roles to load the parts.

Be sure that you have a backup of the clear key parts. This is commonly done by recording the values on paper and independently storing them in safes.

- ▶ Randomly generate a master key.

There are two ways to randomly generate a master key:

- Within the coprocessor

You can request that the master key value be randomly generated within the coprocessor. Using this method, no one can know the value of the master key because the key value is not available outside the coprocessor. To back up the master key, you must clone it to another coprocessor.

- Outside the coprocessor

Within the Web-based configuration utility, you can request that random values be generated for your key parts. The generated values will appear in your window. You should write down these values so that you can store them as a backup.

- ▶ Clone a master key.

The cryptographic coprocessors have the ability to copy a master key from one coprocessor to another without exposing the master key value. This is done through a process called *cloning*. The master key is split into  $n$  shares, of which  $m$  shares ( $m \leq n$ ) can reconstitute the master key value. Any combination of shares fewer than  $m$  cannot reproduce the master key value. One or more shares are moved separately to the target coprocessor, and when enough shares have been loaded, the master key value is reformed. Each master key share is signed and encrypted to ensure secure transport.

The cloning process involves a sender, certifier, and receiver. Many steps need to be performed to clone the master key from one coprocessor to another one. Detailed information about how to perform master key cloning is documented in the Redbooks publication *iSeries Wired Security, Protecting Data over the Network*, SG24-6168.

### 5.2.3 Key tokens

CCA handles keys in a structure called a *CCA key token*. A CCA key token is a data structure that contains information about a key. Use the `Key-Token_Build` verb to build a key token. Use the `Key-Token_Parse` verb to parse a key token.

There are three kinds of key tokens:

► Internal key token

An internal key token is bound to the cryptographic hardware that forms it because the key value is encrypted under the coprocessor's master key. Most CCA operations, such as encrypt, MAC, and sign, require that the key be in an internal key token. An internal key token contains:

- The key value encrypted with a CCA master key
- The control vector
- The master key verification pattern (used to detect under which master key register the key is encrypted)
- The modulus and public key exponent for RSA keys
- A token validation value (a type of checksum to ensure the integrity of the token)

► External key token

An external key token is used to communicate a key between nodes, or to hold a key in a form not enciphered by a CCA master key. An external key token contains:

- The key value encrypted with a CCA transport key (a KEK used only for exporting or importing)
- The control vector
- The modulus and public key exponent for RSA keys
- A token validation value

RSA public keys are never encrypted. When they are not accompanied by the private key, they are stored as an external key token.

► Null key token

A null key token has no key value, but has a control vector.

### 5.2.4 Keystore files

CCA keystore files are DB2 files. You can create a keystore file through the Web-based configuration utility, or your application program can create one using the `Key_Storage_Initialization` verb.

**Note:** If you specify a keystore file that already exists, it will be deleted and recreated without any warning.



You can create two different types of keystores. The Cryptographic Coprocessor uses one type to store symmetric (DES and double-length (112-bit) TDES) keys and the other to store asymmetric (RSA) keys.

You can create as many keystore files of either type as you wish. A CCA keystore file is used to store both data-encrypting keys and key-encrypting keys.

**Note:** Two prototype files, QAC6KEYST (for the DES keystore) and QAC6PKEYST (for PKA keystore), are shipped in the QCCA library. Do not delete, initialize, or alter these files in any way.

Securing your keystore files with object authorities is important. Refer to 5.4, “Establishing a secure keystore environment” on page 65, for help.

When working with keys from keystore you must first designate a keystore file. Do this using the `Key_Storage_Designate` verb, or by defaulting to the keystore file configured in the cryptographic device description.

To generate a key in keystore, first use the `DES_Key_Record_Create` or the `PKA_Key_Record_Create` verb. These verbs add a key record with a null key token to a keystore file. Then generate a key into the key record with the `Key_Generate` or `PKA_Key_Generate` verb.

DES keystore records hold internal key tokens. PKA keystore records can hold internal or external key tokens.

You can also import keys into keystore with the `Clear_Key_Import`, `Data_Key_Import`, `Multiple_Clear_Key_Import`, or `PKA_Key_Import` verbs.

Key records are referenced by key label. A key label can hold one to seven name tokens separated with a period (.). Each name token can be one to eight characters in length, for a maximum total length of 64 characters.

## 5.2.5 Retain keys

RSA keys can be retained in the hardware secure module. To load an RSA key into retain storage you use the `PKA_Public_Key_Hash` or `PKA_Public_Key_Register` verb. To generate a PKA key in retain storage use the `PKA_Key_Generate` verb. In this case, only the public key is returned.

You reference a retained key by label. Depending on how the coprocessor storage is being used, normally you can store between 75 and 150 RSA keys in retain storage.

## 5.2.6 Control vectors

CCA supports a sophisticated control vector set. Refer to 3.5.2, “Key use” on page 31, for a general description of control vectors. A CCA control vector defines the key type and key usage.

- ▶ The key type determines with what verbs a key can be used. Key type might allow a key to be used with several verbs, or it might restrict the key to one verb. For example, a DES key defined as type `MACVER` can only be used with the `MAC_Verify` verb. Whereas, a DES key defined as type `DATA` can be used with the `Encipher`, `Decipher`, `MAC_Generate`, or `MAC_Verify` verbs.

- ▶ Key usage values further restrict the use of the key. For example, you can allow or disallow a key to be exported, you can force the key to be a double-length key, and you can enforce what type or form of key a KEK can act on. For example, a KEK defined as type EXPORTER can only be specified on verbs that encrypt a key for export. But, if the usage is defined as NOT\_KEK, it cannot be used to export a key that is a KEK.

A CCA control vector can be applied to keys encrypted under a master key or encrypted under a KEK. The CCA control vector makes it possible to give symmetric keys asymmetric properties. For example, the same key value can be limited to encryption only or decryption only, generating a MAC or verifying a MAC, importing a key or exporting a key, generating a PIN or verifying a PIN. In general, using control vectors that reduce the capability of the key will reduce the possibilities for misuse.

### 5.2.7 Key identifier

Many CCA verbs support a *Key identifier* parameter. The Key identifier parameter must contain a variable holding a key token or a key label. The key label is an indirect reference to a key token record in a keystore file.

### 5.2.8 Key distribution

This section discusses the distribution of keys.

#### **Between CCA nodes**

This section discusses key distribution between CCA nodes.

#### ***Using a symmetric KEK***

To move a key securely between nodes, encrypt it with a symmetric KEK called a transport key. The transport key must be installed at both locations.

On the sending side, the control vector for the transport key defines it as an exporter key (that is, it can translate a key from an internal form (encrypted under a CCA master key) to an external form (encrypted under the transport key). On the receiving end, the control vector for the transport key defines it as an importer key (that is, it can translate the key from external form to internal form). This establishes a one-way key distribution channel, as depicted in Figure 5-4.

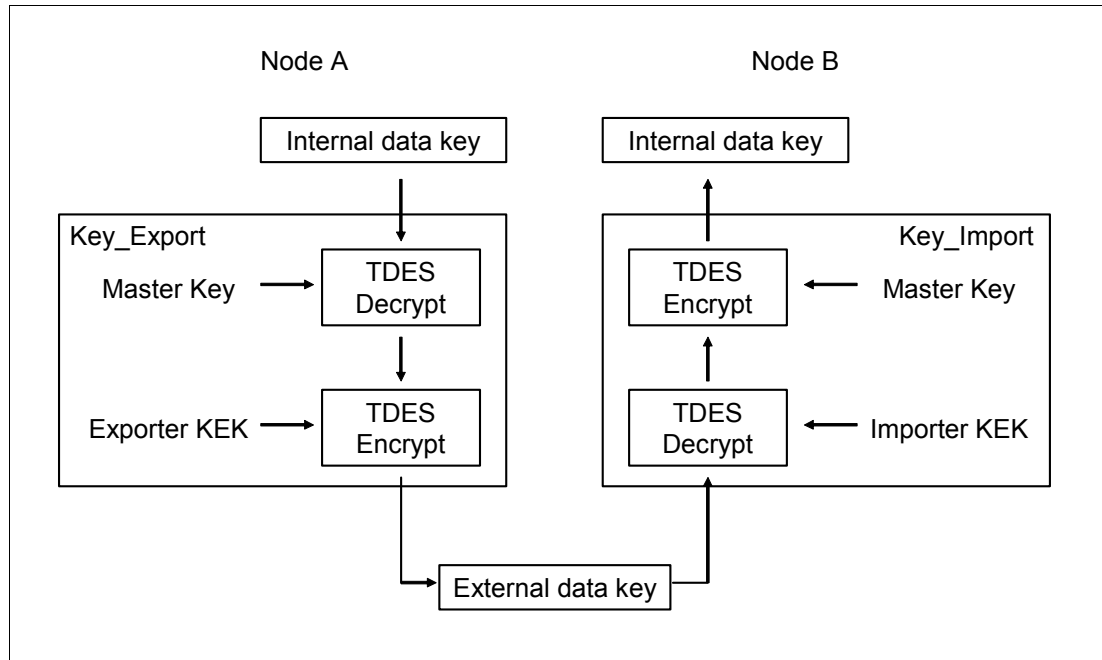


Figure 5-4 One-way key distribution channel

### Using a PKA KEK

Use the `PKA_Symmetric_Key_Export` verb to encrypt a symmetric key under the RSA public key belonging to the intended key recipient. The verb exports the key by decrypting it with the master key and then encrypting it with the RSA public key.

Use the `PKA_Symmetric_Key_Import` verb to recover a symmetric key encrypted under an RSA public key. The verb imports the key by decrypting it with the RSA private key and then encrypting with the master key.

**Note:** Be careful enabling verbs that allow key export. If a key does not need to be exported, then set the control vector such that it will not allow it.

### Between a CCA and non-CCA node

To move a key between a CCA system and a non-CCA system, you must use a special technique called the *Pre-exclusive-OR* technique. Basically, on the CCA side, the transport KEK that will be used to export or import the key must be XORed with the key's control vector. This removes the influence of the control vector on the KEK, so that the key will be translated correctly when exported or imported. This process is described in more detail in Appendix C, "CCA control-vector definitions and key encryption," in the *CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors*.

## Between two System i's

Another method to distribute CCA keys from one System i to another is to move the entire keystore file. The master keys on the coprocessors must be set up identically. Many steps need to be performed to clone the master key from one coprocessor to another one. Detailed information about how to perform master key cloning is documented in the Redbooks publication *iSeries Wired Security, Protecting Data over the Network*, SG24-6168.

### 5.2.9 Changing master keys

When you set a master key you should translate all keys that were encrypted under the former master key to avoid losing access to them. You can translate keys in keystore by using the Cryptographic Coprocessor configuration Web-based utility or by using the `Key_Token_Change` or `PKA_Key_Token_Change` verbs. If you do not translate your keys after a master key is changed and the master key is changed again, you will lose all keys encrypted under that master key and all data encrypted under those keys.

### 5.2.10 Generating keys

CCA uses its cryptographically secure random number generator to generate:

- ▶ Clear keys

When generating a clear symmetric key, use the odd-parity mode of the `Random_Number_Generate` verb. Use the `Clear_Key_Import` or `Multiple_Clear_Key_Import` verb to create a key token in application storage or in a keystore file with the key value encrypted under the master key.

When generating a clear RSA key pair, use the `PKA_Key_Generate` verb.

- ▶ Clear key parts

When generating symmetric key parts, use the odd-parity mode of the `Random_Number_Generate` verb for one key part and the even-parity mode for all other key parts. This will ensure that the key will end up with odd parity. The key parts should be owned by different individuals. The `Key_Part_Import` verb is used to load the key parts one at a time. The key parts are XORed together to form a symmetric key.

- ▶ Encrypted internal keys

The `Key_Generate` and `PKA_Key_Generate` verbs return an internal key token to the application program or to a record in a keystore file. The key value is encrypted under the master key and is ready for use.

- ▶ Encrypted external keys

Besides returning the generated key encrypted under the master key, the `Key_Generate` and `PKA_Key_Generate` verbs can return the generated key encrypted under an exporter key (a KEK used for exporting) or an importer key (a KEK used for importing) in external key tokens.

The `PKA_Symmetric_Key_Generate` verb generates a key encrypted under an RSA public key for distribution to another node (that has the corresponding private key). The generated key is also returned encrypted under the symmetric master key or a DES KEK.

- ▶ Encrypted key pairs

The `Key_Generate` verb allows you to generate an encrypted key that is output twice under opposite control vectors, for example, one with encipher and one with decipher capabilities.

## 5.2.11 Backing up keys

In this section we discuss backing up keys.

### Master keys

There are basically two ways to back up CCA master keys:

- ▶ Save the key parts (for example, on paper) and store them in a secure place, such as in separate lock boxes.
- ▶ Clone the master key to another coprocessor. Many steps need to be performed to clone the master key from one coprocessor to another one. Detailed information about how to perform master key cloning is documented in the Redbooks publication *iSeries Wired Security, Protecting Data over the Network*, SG24-6168.

### Keystore files

CCA keystore files are database files. You can back them up using i5/OS save/restore functions such as the SAVOBJ and SAVLIB CL commands.

Any time a new key is added to a keystore file, you should make a backup of the file. In addition, any time the keystore file is translated, you need to make a new backup of the file.

Do not forget about keys stored outside of keystore. These should be backed up as well. Generally, these should not be encrypted under a master key. If you store a key outside of keystore, it is best to encrypt it under a KEK, such as an importer key. If you encrypt it under a master key and that master key is changed, you must remember to translate the key.

## 5.2.12 Using multiple coprocessors

System i supports multiple coprocessors. Depending on your System i model and which coprocessors you own, it can support from 3 to 32 coprocessors on the system, and from 3 to 8 coprocessors per partition. Refer to the iSeries Information Center at the following URL for specifics:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/topic/rzajc/rzajcco4758.htm>

During System i IPL, all detected cryptographic coprocessors are assigned a resource name beginning with CRP01, then CRP02, and so on. To use a coprocessor, you must create a device description object and specify one of the cryptographic resource names.

Spreading the work across multiple Cryptographic Coprocessors and multiple jobs gives you better performance provided that they are all configured the same. However, there may be situations when you do not want all of the coprocessors to be configured the same. You may even want groups of coprocessors with different configurations.

There are a number of considerations when using multiple coprocessors:

- ▶ Your application can use the default coprocessor or it can explicitly select a coprocessor with the Cryptographic\_Resource\_Allocate verb. On this verb you specify the device description object name, not the resource name. The default device description object name is always "CRP01". Once the coprocessor is allocated, CCA requests are routed to it until it is deallocated. To deallocate an allocated coprocessor, use the Cryptographic\_Resource\_Deallocate verb. These verbs are scoped to the process, and all threads within the process use the same coprocessor.
- ▶ If you configure all of the coprocessors the same, then all internal keys will work identically on all of the coprocessors. Any data encrypted on one coprocessor can be decrypted on a

different coprocessor. All keystore files will work interchangeably with any of the coprocessors.

The most important part of configuring the coprocessors identically is the master keys. If you entered the master key in parts for one coprocessor, you must enter the same master key parts for all of the other coprocessors if you want them to work interchangeably. If a random master key was generated inside of the coprocessor, then you must clone the master key to the other coprocessors if you want all of the coprocessors to work interchangeably.

- ▶ If you configure your coprocessors differently, you must keep track of which keystore files and operational keys will work for a given coprocessor. While configuring the coprocessors differently may limit the scalability of cryptographic applications, it can provide more granularity in terms of security. For example, you can grant different object authorities to different cryptographic device descriptions.

If you use retained PKA keys then the coprocessors are not interchangeable. Retained keys cannot be exported outside of the coprocessor. Therefore, any cryptographic request that uses a retained key must be sent to the coprocessor that stores the retained key.

## 5.3 Roll your own key management

You may wish to develop your own key management. For example, SQL does not provide any sort of management for your secret passwords (which are essentially keys). Some considerations when developing your own key management on System i are:

- ▶ How are your keys generated?

If you are generating passwords, it is best to use a 128-character password to ensure a 128-bit security level. If you are generating actual keys, use a cryptographically secure random or pseudorandom number generator. For example, use the Cryptographic Services Generate Pseudorandom Numbers API.

Refer to 3.3.1, “Generating a key value” on page 28, for further discussion.

- ▶ Where will you store your keys?

Keys (or passwords) should not be hardcoded. If you store a key on the system, it should be stored in a System i object, such as a database file, a stream file, or a data area.

- ▶ How will your application access your keys?

Your application must have a means of identifying a particular key. Usually, this is done by assigning a label.

When your application retrieves a key, it should only retrieve the one called for and no others.

- ▶ How are your object authorities set?

Perhaps the most important consideration is how you set up object authorities to both your keystore objects and to the application that accesses the keystore objects. Refer to 5.4, “Establishing a secure keystore environment” on page 65, for an in-depth discussion.

Can keys be authorized individually? One way to accomplish this is by allowing the creation of multiple keystore objects.

- ▶ Will the keys be stored encrypted?

If you move your keystore object off the system (for example, via a SAVOBJ for backup purposes), the keys should be encrypted. If you also are concerned that an unauthorized person may obtain access to your keystore object on the system, then encrypt the keys as

soon as they are created. Otherwise, you can encrypt the keys prior to them leaving the system.

The question now is about the management of the KEK.

- Who will be in charge of managing the KEK? How will it be established? Should it be established in parts?
- How is the KEK protected?

Perhaps the safest method is to not store the KEK on the system at all. For example, you could establish it interactively at job startup.

An easier method is to use a Cryptographic Services master key. You cannot use the master key to encrypt your keys directly, but you can establish a KEK under the master key that you could use to encrypt your keys.

- Will you be changing the KEK, and how often?

When the KEK is changed, the keys must be translated. To do so:

- i. Create a second KEK.
- ii. Translate your keys from the old KEK to the new KEK. For example, you could use the Cryptographic Services Translate Data API.

- ▶ When do you back up your keys?

Your keys should be backed up whenever you add a key or change a key. If the keys are encrypted, they should also be backed up whenever they are translated. Do not forget to back up the KEK as well.

## 5.4 Establishing a secure keystore environment

A keystore file is an object that contains encryption keys. Although the keys inside the keystore are themselves maintained in an encrypted state, for both CCA and Cryptographic Services, the keystore is a simple physical file object. For this reason, we recommend that you still take measures to ensure that the store is protected from misuse.

Although i5/OS object-level security is outside the scope of this book, we feel that the importance of this topic warrants a quick tour of one way to establish a secure environment for a keystore file.

More detailed and expansive information about i5/OS object-level security and auditing can be found in the *iSeries Security Reference*, SC41-5302-09, at:

<http://publib.boulder.ibm.com/infocenter/series/v5r4/topic/rzahg/rzahgsecref.htm>

The following explanations are written in the context of using a Cryptographic Services keystore file, but apply equally to CCA keystore or other i5/OS objects containing keys.

**Note:** No object can be protected from a user with All Object special authority. This also applies if the user is a member of a group (profile) that has \*ALLOBJ.

### 5.4.1 Object security 101

The i5/OS security algorithm requires that a user requesting access to an object must first have authority to the library that it resides in.

To access the contents of a library, the user must have (at least) operational (OPR) object rights along with READ and EXECUTE data rights to the library. These rights are most commonly provided via the object use (\*USE) authority template.

Object rights can be viewed and edited with the Edit Object Authority (EDTOBJAUT) CL command or via iSeries Navigator.

Once the user has the necessary authority to the library, most actions on objects inside that library are controlled by the objects' own authorities.

**Note:** If a library exists in the user's library list, and the user refers to an object in that library without the library designation (that is, using library \*LIBL), then i5/OS does *not* check the authority to the library unless the action requested affects the library itself.

## 5.4.2 Create user profiles

When securing an object, it is often desirable for the object to be owned and accessed by user profiles created specifically for that task.

Although it is common to combine the ownership and access function to a single user profile, we generally do not recommend this, as an object owner has certain authorities to the object just by owning it. Also, making a user profile a member of the owner as a group profile provides that same ownership right to each member.

### Keystore owner

To create a user profile suitable for owning the keystore, we need a basic user with no special capabilities and no ability to actually sign onto the system:

```
CRTUSRPRF USRPRF(KEYOWNR) PASSWORD(*NONE) STATUS(*DISABLED) INLPGM(*NONE)
INLMNU(*SIGNOFF) LMTCPB(*YES) TEXT('Keystore Owner') SPCAUT(*NONE)
```

### Keystore user

To create a user profile to access the keystore, we need another basic user similar to the one created for the keystore owner:

```
CRTUSRPRF USRPRF(KEYUSR) PASSWORD(*NONE) STATUS(*DISABLED) INLPGM(*NONE)
INLMNU(*SIGNOFF) LMTCPB(*YES) TEXT('Keystore User') SPCAUT(*NONE)
```

## 5.4.3 Location, location, location

The first step in securing the keystore file is to create a secure library to place it in. This can be accomplished using the Create Library (CRTLIB) CL command.

An example of creating a library that has public authority of \*EXCLUDE is:

```
CRTLIB LIB(KEYSTORLIB) AUT(*EXCLUDE) CRTAUT(*EXCLUDE) TEXT('Secure Library')
```

Generally, the user who creates the library will be designated as the owner of that library. By default, this provides that user with \*ALL rights to the library. It is desirable to change this to an inactive user profile to reduce any vulnerability. We accomplish this with the Change Object Owner (CHGOBJOWN) CL command, as below:

```
CHGOBJOWN OBJ(KEYSTORLIB) OBJTYPE(*LIB) NEWOWN(KEYOWNR) CUROWNAUT(*REVOKE)
```



As the owner, KEYOWNER is automatically given the authority of \*ALL. Although ownership alone affords distinct privileges (including the ability to authorize themselves and others to the object), we revoke the private authority for this library from this user for clarity:

```
RVKOBJAUT OBJ(KEYSTORLIB) OBJTYPE(*LIB) USER(KEYOWNER) AUT(*ALL)
```

At this point, only those profiles with \*ALLOBJ special authority can perform any action on the library and have the possibility of accessing the objects that the library will contain. In order to allow the profile KEYUSR access to the library, we need to specifically authorize it. This profile still only requires minimal access to the library as follows:

```
GRTOBJAUT OBJ(KEYSTORLIB) OBJTYPE(*LIB) USER(KEYUSR) AUT(*USE)
```

This will permit access to any existing objects inside the library. KEYUSR will not be able to add new objects into the library, including new members to an existing physical file. The actions that can be performed on the objects are defined by the objects themselves. Understand that \*USE authority to the library does not prevent changes or deletions of objects if the user has that level of access to the objects themselves.

Now we are ready to work at the object level.

#### 5.4.4 Secure the keystore

Once the keystore is created, it is important to define the authorities that users have. This is because if a user has access to a library, most activities are controlled at the object level.

We previously defined our library to be secured from everyone except KEYUSR, so this next step is just one extra level of protection. Here we ensure that unlisted users do not have any direct authority to the keystore:

```
GRTOBJAUT OBJ(KEYSTORLIB/keystorefile) OBJTYPE(*FILE) USER(*PUBLIC) AUT(*EXCLUDE)
```

Next, we transfer the ownership to the KEYOWNER profile and then, for clarity only, revoke the private authority that they automatically gained with this transfer:

```
CHGOBJOWN OBJ(KEYSTORLIB/keystorefile) OBJTYPE(*FILE) NEWOWN(KEYOWNER)  
CUROWNAUT(*REVOKE)
```

```
RVKOBJAUT OBJ(KEYSTORLIB/keystorefile) OBJTYPE(*FILE) USER(KEYOWNER) AUT(*ALL)
```

Finally, we authorize KEYUSR to all data functions (add/change/delete) for the keystore file:

```
GRTOBJAUT OBJ(KEYSTORLIB/keystorefile) OBJTYPE(*FILE) USER(KEYUSR) AUT(*CHANGE)
```

**Note:** You may want to consider creating an additional profile (for example, KEYADMIN) that has \*USE access to the library and \*CHANGE access to the keystore file. This profile would then be used to perform all key maintenance functions using the same programming methodologies outlined in this section for KEYUSR.

This would enable us to grant the KEYUSR profile \*USE authority (read-only) to the keystore file.

## 5.4.5 Accessing the keystore

In our scenario, only KEYUSR currently has any access to the keystore, and that profile cannot sign on. When an application program needs to decrypt data it also needs to have authority to access the contents.

There are two main methodologies to accomplish this: switch profile APIs and adopted authority.

### Switch profile APIs

These programming interfaces allow a job to switch midstream to another profile and gain the rights of that profile. By switching to the user profile KEYUSR the program can temporarily gain access to the keystore.

Although this is a more modern approach, the process requires more explanation than is possible in this book.

Also, be careful not to provide access to a command line while the profile switch is in effect, as that command line will now provide direct access to the KEYSTORLIB library and the *keystorefile* object contained within it.

### Adopted authority

By default, when an application program runs, it only has the authority of the user that is running it. However, it is possible to define the program attributes to instruct the system that if the user's authority proves insufficient, then the program *owner's* authority should be checked. This function is called adopting authority.

Using the infrastructure that we created in this section, we can now use a program to access the keystore using the adopted authority of KEYUSR. The creation of this program is not covered in this section, but the commands to set a program called *dcryptpgm* in library appllib to run with KEYUSR's adopted authority are noted below.

First, we need to change the program to be owned by the KEYUSR profile:

```
CHGOBJOWN OBJ(applib/dcryptpgm) OBJTYPE(*PGM) NEWOWN(KEYUSR) CUROWNOUT(*REVOKE)
```

Then set the program to defer to the owner's authority if the user's authority proves insufficient:

```
CHGPGM PGM(applib/dcryptpgm) USRPRF(*OWNER)
```

As we have previously authorized the profile KEYUSR to the library and to the keystore file, any user with at least \*USE authority to the *dcryptdta* program will now be able to perform the necessary encryption/decryption functions. However, if the same user attempts to access the keystore file directly, he will be denied access.

**Note:** If using adopted authority, we recommend considering using the Use Adopted Authority (QUSEADPAUT) system value. This control accepts the name of an authorization list that may contain a list of user profiles. This list specifies which users can create a program that adopts authority.

## 5.4.6 Auditing keystore access

With a file as sensitive and critical to system data as a keystore, you might want to consider auditing. Fortunately, i5/OS contains the functionality to easily accomplish this.

### Auditing infrastructure

It is first necessary to create a security audit journal if one does not exist. After that we use a combination of system values and profile/object settings to define what we want to have audited.

Use the Display Security Auditing (DSPSECAUD) CL command to determine whether a security audit journal already exists and what types of system-information is currently being audited.

If auditing is not active, use the Change Security Auditing (CHGSECAUD) CL command to perform all of the necessary steps including creating the journal and journal receiver and setting the QAUDCTL and QAUDLVL/2 system values.

Ensure that the QAUDCTL system value includes the \*OBJAUD option if you wish to perform object-level auditing.

### Object-level auditing

i5/OS auditing can be performed on individual objects. To accomplish this, the auditing control system value (QAUDCTL) needs to include the value \*OBJAUD. From there, it is a simple matter of activating the desired level of auditing:

```
CHGOBJAUD OBJ(KEYSTORLIB/keystorefile) OBJTYPE(*FILE) OBJAUD(*ALL|*CHANGE)
```

An Object Auditing (OBJAUD) value of \*ALL will audit all change and use activities. A value of \*CHANGE will only audit change activities. Audit journal entries of 'ZR' and 'ZC' will be recorded if the object is read or changed, respectively.

Another option is to audit a program designed to accesses the keystore. The process is the same for programs as for files. Understand that this option alone will not record accesses made by unofficial programs. However, by combining the two approaches, it is possible (with some programming) to determine whether an unofficial program is gaining access to the keystore.

### System-level auditing

i5/OS is also capable of auditing activities involving program authority adoptions. By including the \*PGMADP value in the auditing level system value (QAUDLVL), the system will write an 'AP' entry to the audit journal every time a program that adopts authority is used.

Finally, a general auditing recommendation is to include the \*AUTFAIL option in QAUDLVL. This option writes an 'AF' entry to the audit journal whenever an action fails due to insufficient authority (which would include a user attempting to directly access the secured keystore).

### Reporting

To interrogate the audit journal entries, use the Display Audit Journal (DSPAUDJRNE) CL command or a commercial application designed for this task.





## Choosing a data encryption method

This chapter provides guidance about choosing a cryptographic interface, a cryptographic algorithm, and a mode of operation. It also provides additional tips and techniques.

## 6.1 Factors to consider

The following factors should be considered when choosing a method for your encryption project:

- ▶ What algorithms are supported by the cryptographic interface that you wish to use? For example, your compliance requirements may require you to store master keys on tamper-resistant hardware, in which case you will use the CCA API set for the cryptographic coprocessors, which restricts you to DES, TDES, and RSA.
- ▶ Are you required to be compatible with another cryptographic application? For example, will you be decrypting data that was encrypted elsewhere?
- ▶ Encryption often requires the plaintext data to be padded before encrypting. Can you handle ciphertext that is longer than the plaintext, or must it be the same length? For example, when encrypting DB2 data, you may not be able to change the width of your columns.

## 6.2 Choosing an interface

Several cryptographic API sets are available to i5/OS applications for encrypting data at rest. Refer to 2.3.2, “Cryptographic interfaces” on page 23, for a short description of each. Table 6-1 compares System i cryptographic interfaces.

Table 6-1 V5R4 Cryptographic interfaces comparison

	i5/OS Cryptographic Services	Common Crypto Architecture	SQL built-ins	Java Cryptography
<b>Use</b>	▶ General purpose	▶ General purpose ▶ Financial	▶ DB2 column encryption	▶ General purpose
<b>Cryptographic service provider</b>	▶ i5/OS LIC ▶ 2058 <sup>a</sup>	▶ 4758 <sup>b</sup> ▶ 4764	▶ i5/OS LIC	▶ Java Crypto Extension
<b>Supported algorithms</b>	▶ DES ▶ TDES ▶ AES ▶ RC2 ▶ RC4 ▶ RSA ▶ D-H ▶ MD5 ▶ SHA1 ▶ SHA2	▶ DES ▶ TDES ▶ RSA ▶ MDC ▶ MD5 ▶ SHA1 ▶ RIPEMD-160	▶ RC2 ▶ TDES	▶ DES ▶ TDES ▶ AES ▶ RC2 ▶ MARS ▶ Blowfish ▶ RC4 ▶ SEAL ▶ RSA ▶ D-H ▶ DSA ▶ MD2 ▶ MD5 ▶ SHA1 ▶ SHA2
<b>Key management</b>	▶ MK protected KS <sup>c</sup> ▶ MKs stored in LIC	▶ MK protected KS ▶ MKs stored on secure hardware	▶ No key management	▶ PW protected keystore <sup>d</sup>
<b>Pre-req's<sup>e</sup></b>	▶	▶ SS1 Option 35 ▶ 5733CY1 (4764)	▶	▶ 5722JV1 JDK™
<b>Cost</b>	▶ Part of i5/OS	▶ \$\$ for coprocessor	▶ Part of i5/OS	▶ Part of i5/OS

a. The 2058 Cryptographic Accelerator is no longer available. However, if you have one, it is still supported.

b. The 4758 Cryptographic Coprocessor is no longer available. However, if you have one, it is still supported.

c. Master key protected keystore.

d. Password protected keystore.

e. Prior to V5R4, 5722-AC3 Cryptographic Service Provider, a no-charge program product, is required to enable cryptography on the system. In V5R4, this product is no longer required.

## 6.3 Choosing an algorithm

For a general discussion on cryptographic algorithms, refer to 2.1, “Cryptographic algorithms” on page 12. For a discussion on key strengths, refer to 3.2, “Size matters” on page 26.

### 6.3.1 Cipher algorithm

In this section we discuss our recommendations for using a cipher algorithm.

## DES

NIST withdrew DES as a standard in 2005. Because of its short key length, it is no longer considered secure against the power of today's processors. In fact, DES keys have been broken in under 24 hours. Do not use DES.

## AES

AES is the official replacement for DES. AES is an excellent choice both for security and speed. AES is available through the Cryptographic Services APIs and Java Cryptography.

AES supports three key lengths: 128 bits (16 bytes), 192 bits (24 bytes), and 256 bits (32 bytes). It is best to use a 256-bit key. Because of collision attacks, it actually gives you 128-bit security.

Use CBC mode to hide patterns in the data. If the size of your data is not on a block size boundary, you have the following options.

- ▶ Select the pad option. This increases the size of your ciphertext to the next block-size multiple.
- ▶ To keep your ciphertext the same size as your plaintext, implement CUSP mode of operation. CUSP mode can be implemented as follows:
  - a. Use CBC mode to encrypt up to the last partial block of plaintext.
  - b. Encrypt the last block of ciphertext again.
  - c. XOR the last partial block of plaintext data with the same number of bytes from step b.

If the data length is less than the block size, the initialization vector is used in place of the last block of ciphertext in step a. This process is identical for encrypting or decrypting.

Cryptographic Services supports three AES block sizes: 128 bits (16 bytes), 192 bits (24 bytes), and 256 bits (32 bytes). A 256-bit block size is best because a larger block size reduces the number of ciphertext collisions that leak information. However, the AES standard only specifies a 128-bit block size. Therefore, when using a 256-bit key with a 128-bit block size, it is important to limit how much total data is encrypted with a single key. (Otherwise, you might as well use a smaller key size.) With CBC mode, you should limit it to 232 blocks or so.

## TDES

If AES is not a choice, use triple DES. Three-key TDES is preferable over two-key TDES.

Use CBC mode to hide patterns in the data. If your data is not on a block size boundary, you have the following options:

- ▶ Select the pad option. This increases the size of your ciphertext to the next 8-byte boundary.
- ▶ Implement CUSP mode, as explained above, for the last partial data block. This keeps your ciphertext the same size as your plaintext.
- ▶ With Cryptographic Services, you can also use OFB or CFB modes to keep your ciphertext the same size as your plaintext. Even though these modes use TDES like a stream cipher, Cryptographic Services requires the length of your data to be a multiple of eight. Therefore, perform the following steps:
  - a. Pad out your data (for example, with nulls) to the next 8-byte block boundary.
  - b. Encrypt using OFB or CFB mode. We prefer CFB mode because with OFB mode the plaintext is easy to manipulate via changes to the ciphertext.
  - c. Trim off the end of the ciphertext the number of bytes that was padded.



Decryption is similar. You must pad out the ciphertext (for example, with nulls) to the next 8-byte boundary and then decrypt.

**Note:** When reusing a key with OFB or CFB modes, you must *never* reuse an IV or the security will be severely compromised.

## RC2

Although RC2 has been around since 1987, the details of the algorithm were kept secret until 1996. There does not appear to be as much analysis of RC2 as there is for TDES and AES. We believe that it is preferable to use AES or TDES over RC2.

## RC4

RC4 is a stream cipher and is available through Cryptographic Services and Java Cryptography. One of the attractions of RC4 is that the ciphertext is always equal in length to the plaintext. However, key management can be a problem. Because of the nature of the RC4-compatible algorithm, using the same key for more than one message severely compromises security. If keeping the ciphertext length equal to the plaintext length is a requirement, we prefer that you use a block cipher with CTR or CFB modes of operation, rather than RC4.

### 6.3.2 Hash and HMAC algorithms

Use a strong hash algorithm like SHA-256 or SHA-512 if possible. (Do not bother using SHA-384. The underlying function performs all the work of SHA-512 and then throws away part of the results.) Because of weaknesses in the SHA hash functions, it is best to perform a double hash (that is, hash the data then hash the result). Do not use MD5.

We recommend SHA-256 HMAC for your authentication function, as well. Use all 256 bits (32 bytes) for your MAC value if possible. A double hash is not needed for HMAC operations.

## 6.4 Tips and techniques

Review the following tips and techniques:

- ▶ Do not encrypt and store ciphertext when all that is needed is a hash. In many situations, the application program does not actually need to recover plaintext data from encrypted data. For example, an application that needs to verify a credit card holder or look up information related to a credit card need not store the encrypted credit card data, but rather a hash of the data.
- ▶ One of the biggest misconceptions in cryptography is that encrypting data will prevent the alteration of data. However, just because data is encrypted does not mean that it cannot be deleted, repeated, or even altered. Even when using modes of operation, a bit of altered ciphertext affects at most two blocks of plaintext, and it is often difficult for the application to detect it. Therefore, it is almost always a good idea to apply an authentication function to your data. Generating and verifying a MAC using an HMAC function has better performance than using a block cipher. Again, we recommend SHA-256 HMAC.
- ▶ It can be argued that authentication of data is more important than concealing data. For that reason, it is best to apply an authentication function first, and then encrypt the data so that the MAC is also concealed. Do not use the same key for encryption and authentication.

- ▶ Simple attacks can be performed on the MAC of a plaintext message. To prevent these attacks, concatenate the length of the input data to the start of the message, then MAC the entire string.
- ▶ When using a block cipher, to avoid identical ciphertext in the first block, you must use a substantially different IV for each message that uses the same key. One option is to generate a random value to use as the IV. Or you can use a nonce (a unique number), but you must first encrypt it before using it as an IV, and you must ensure that for a given key, the counter never wraps. Note that the IV need not be kept secret.
- ▶ Encrypt only sensitive information. Non-sensitive information is often easily accessed and allows an adversary who has access to both the plaintext and the ciphertext to mount a *known plaintext attack*. For example, an adversary knows the standard header or trailer of your records and can see the encrypted records. Therefore, you should not encrypt the beginning and ending of the record. Or perhaps he knows that you store a last name at the start of an encrypted data record and has access to it and also to a plaintext list of those last names. Even more dangerous would be to allow the adversary to add his own last name, which he then can observe encrypted. This is known as a *chosen plaintext attack*.



## Database considerations

Working with encrypted data in DB2 for i5/OS presents unique challenges when compared to traditional plaintext data. These challenges include the impact on both new and existing database file structures, as well as the applications used to access them.

How you design any database influences the efficiency, as well as the availability, of the applications that access it. This especially holds true when encrypted data is involved.

This chapter discusses the impact that hosting encrypted data has on i5/OS database design, how you move to an encrypted data model, and also how encrypting your data affects some popular utility applications commonly used to add, update, and view file data.

## 7.1 Understanding how the database is used

When planning a database change, whether it is in preparation for encryption or as a result of some other requirement, it is critical that the use of the database be documented and understood.

If field attributes (data type and length) need to be modified to accommodate ciphertext data, all references to that field need to be located and evaluated for the impact of the modification.

Most high-level language (HLL) program languages, such as RPG or COBOL, address fields differently depending on their data type—an attribute commonly affected by encryption. This dictates that all programs referencing the field need to be assessed, and potentially modified, over and above any modifications required for the encryption/decryption of the data.

### Discovering application usage

In order to understand how your data is used throughout the application, you must research which files contain those fields, and when and where those files are used by the application. You also may want to differentiate between the uses of the file data (read-only versus read/write).

Although there are commercial applications available specifically to perform this function, it is possible to build a simple map of file usage using the Display Program Reference (DSPPGMREF) CL command. Select the target object type (\*PGM) and direct the command output to a file. Query the results to find which programs refer to the files containing any fields in question.

Each program needs to be reviewed by a programmer for field references that would be affected by:

- ▶ Field length changes
- ▶ Field data type changes (if currently numeric)
- ▶ Encryption/Decryption requirements

**Note:** The DSPPGMREF command only identifies native file declares. If your application utilizes embedded SQL, then you will also need to search the source code for those file references.

## 7.2 How encryption impacts database structure

The storage requirement for ciphertext is often different from that of plaintext data. This difference manifests itself in three ways:

- ▶ Increased data length

The encryption algorithm that you decide to use, as well as the length of the plaintext data, impacts the length of the data in its encrypted form.

- ▶ Alternate character set

The ciphertext version of your data will include character representations that cause decimal-data errors if stored in numeric fields. If the plaintext field is currently defined in the file as a numeric field, then this needs to be changed to a field type that can correctly handle the non-numeric data.

► Specification of CCSID of 65535

CCSID is an abbreviation for Coded Character Set Identifier. It is a 16-bit number that represents a specific encoding of a specific code page. The CCSID value of 65535 indicates that the data is stored in HEX and should not be converted between CCSIDs.

For example, a Social Security number (SSN) in the United States is a unique 9-digit identification number assigned to every legal resident. The data is limited to the characters 0–9. As such, it is entirely possible for that data, in plaintext form, to currently be stored in a field that is defined as packed or zoned numeric.

A Data Description Specification (DDS) file layout example is represented in Figure 7-1.

```

.....A*. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
***** Beginning of data *****
A*
A          R EMPFPR                TEXT('EMPLOYEE RECORD')
A*
A          EMPID                   10A   COLHDG('ID Number')
A          EMPID                   10A   TEXT('ID Number')
A          EMPLNAME                 20A   COLHDG('Last' 'Name')
A          EMPLNAME                 20A   TEXT('Last Name')
A          EMPFNAME                 20A   COLHDG('First' 'Name')
A          EMPFNAME                 20A   TEXT('First Name')
A          EMPSSN                   9P 0  COLHDG('XXX-XX-XXX (SSN)')
A          EMPSSN                   9P 0  TEXT('SSN')
A          EMPDEPT                  10A   COLHDG('Department')
A          EMPDEPT                  10A   TEXT('Department')
A          K EMPID
A*

```

Figure 7-1 Sample employee master file with plaintext Social Security number defined as numeric

However, when this data is encrypted it may become 16 bytes of mixed characters, which can no longer be stored in a numeric field. For example:

```

plaintext: 123456789
ciphertext: Q^a.Wg%A8IluPva#

```

In this case, the field definition needs to be altered to accommodate the additional character combinations as well as the additional field length.

Fortunately, the kind of information that lends itself to being encrypted does not usually require mathematical or date functionality. These fields can therefore be modified to a non-numeric format without many technical challenges. That being said, *any* database modification can have a significant impact on an application that references it.

If a field definition needs to be modified, you need to decide which technique to use for the modification. You have two main options:

- Record layout modification
- Database normalization

These two options are discussed in the following sections.

## 7.2.1 Modifying current record layout structure

The term *record layout modification* refers to a change to the existing field in an existing file. For example, in Figure 7-2 we modified the existing Social Security number field originally shown in Figure 7-1 on page 79, without altering the overall database schema.

```

.....A*. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
***** Beginning of data *****
A*
A          R EMPFPR                TEXT('EMPLOYEE RECORD')
A*
A          EMPID                   10A      COLHDG('ID Number')
A          TEXT('ID Number')
A          EMPLNAME                20A      COLHDG('Last' 'Name')
A          TEXT('Last Name')
A          EMPFNAME                20A      COLHDG('First' 'Name')
A          TEXT('First Name')
A          EMPSSN                  16A      COLHDG('ENCRYPTED SSN')
A          TEXT('ENCRYPTED SSN')
A          CCSID(65535)
A          EMPDEPT                 10A      COLHDG('Department')
A          TEXT('Department')
A          K EMPID
A*

```

Figure 7-2 Employee master file with Social Security number redefined as alpha/numeric

The modification of an existing field requires that the file object be recreated to recognize the new updated definition. If the field requires only a length modification, then there are two approaches that you may consider to regenerate the file while retaining the existing data and file attributes:

- ▶ Change Physical File (CHGPF) CL command

The CHGPF command can be used to regenerate the file object from modified DDS while retaining the integrity of the existing file data.

- ▶ SQL's ALTER TABLE instruction

The ALTER TABLE instruction permits the modification or removal of an existing column (field) as well as the addition of new ones. Although it can be used to alter a file defined by DDS, it is most commonly used to modify the structure of a table created by the CREATE TABLE instruction in SQL. Just remember that if you modify a DDS-based file with this approach, then the DDS source will no longer match the file object.

These two mechanisms are efficient, as they only rebuild the necessary access paths and automatically migrate the data and file attribute information to the new file object. However, if the field requires a data type modification from numeric to alpha/numeric, you are not able to use the CHGPF command or ALTER TABLE instruction. Instead, you must consider how to convert the data to the new format.

**Note:** Some cryptographic algorithms and modes of operation produce cyphertext that is the same length as the plaintext. Refer to Chapter 2, “Algorithms, operations, and System i implementations” on page 11, for more information.

More information about data conversion can be found in 7.3.2, “Encrypting existing data” on page 86.

**Note:** If any programs reference the file via internal definition, then those programs must be located and modified for the new field structure.

## 7.2.2 Normalizing the encrypted fields

As you review the required database changes, you also have an option to externalize the fields that are to be encrypted, into a file of their own. We refer to this process as *database normalization*. For example, instead of altering the original Social Security number field as we did in Figure 7-2 on page 80, we can create an auxiliary file to contain the (encrypted) Social Security number field. We have also included the unique employee ID field from the original employee file, so that we may link the associated records together. This is represented in Figure 7-3.

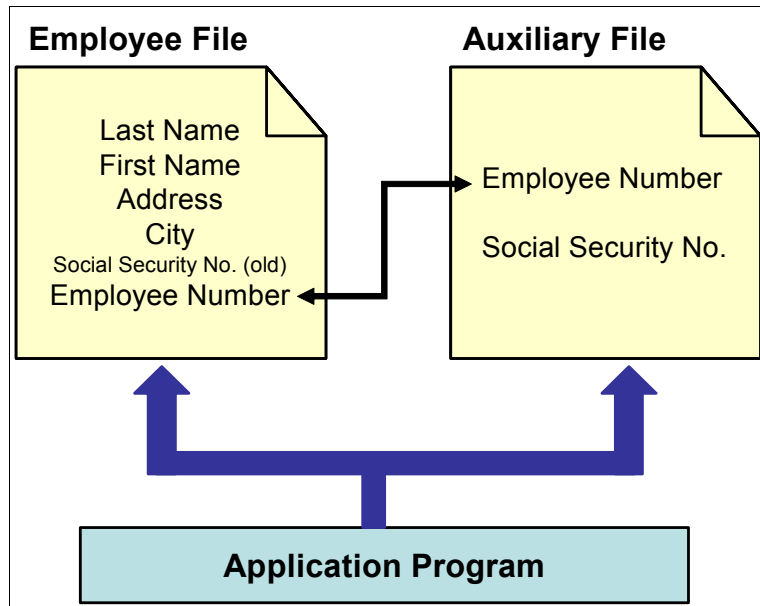


Figure 7-3 Normalizing database structure using an auxiliary file

Normalizing avoids having to recompile the existing file. This may be advantageous to your application environment for reasons that include:

- ▶ The file maintains the same level-check identifiers. Application objects generally need to be created (compiled) in a specific sequence. Database files are created first followed by the application programs that reference them. This is so that the program compiler can retrieve the field structure for reference in the program. If the file is subsequently changed without recompiling the referencing programs, you may receive unexpected results. For this reason, the level-check feature is designed as a warning mechanism to notify the program that the file has been created more recently than the program object. Although the warning can be turned off in the file description, we do not recommend this.
- ▶ Recompiling a file object clears the data stored in the file members. Although there are i5/OS mechanisms (CHGPF command, ALTER TABLE in SQL) to modify a file structure without losing the data, there are restrictions. Data type changes, common in encryption, are included in those restrictions. Not changing the existing structure means not having to

manually migrate the existing data from a temporary location back to the file with the new field format.

- ▶ The application can be modernized in phases. Instead of modifying every program to reference the data from the new location in a single stage, a program could attempt to retrieve and decrypt the ciphertext from the auxiliary file first and, if no match is found, utilize the original plaintext field in the master file. As the data is migrated, the program still handles both scenarios. Although this requires surplus code to be added to the programs, in some cases it might be a legitimate trade-off.

A disadvantage of modification by normalizing the field structure is that you now have to process an additional file I/O for each I/O to the original file. You also end up with obsolete fields in the database. One possible use for a field made obsolete through normalization is to re-populate it with an index to link it to the auxiliary record. This index would be necessary if there is not a unique index in the master file.

**Note:** If the old field is not going to be repopulated with an index to the auxiliary file, then ensure that the plaintext version of the private information is cleared.

Figure 7-4 shows an example of how our auxiliary employee file might look after the Social Security number is normalized.

```

.....A*. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
***** Beginning of data *****
A*
A          R EMPXTR          TEXT('EMPLOYEE EXTENTION')
A          EMPID            10A      COLHDG('ID Number')
A          EMPSSN          16A      TEXT('ID Number')
A          EMPSSN          16A      COLHDG('ENCRYPTED SSN')
A          EMPSSN          16A      TEXT('ENCRYPTED SSN')
A          EMPSSN          16A      CCSID(65535)
A          K EMPID
A*

```

Figure 7-4 Auxiliary employee file for Social Security number

Optionally, by utilizing a join-logical file, or an SQL view, the application can continue to read one file and access all of the data elements of the employee's record.

**Note:** Be aware that join logical files (JLFs) are, by design, read only.

### 7.2.3 Key version field

When reviewing the database structure, you should anticipate the future requirement to change the data encryption key (the key used to convert the data from plaintext to/from ciphertext). This may be driven by a legislative requirement or from a breach of key secrecy. This breach may simply be caused by the key holder leaving the company.

Unlike changing a master key or key encrypting key, the change of a data encrypting key means that all of the data currently encrypted needs to be translated (decrypted and then re-encrypted) using the new key. This can prove to be an even greater undertaking than the original conversion, as we are now requiring the data be decrypted *and* re-encrypted during



the translation. This may not be something that can be performed in a single pass during a dedicated window.

Regardless of whether you chose to modify the existing record format structure or to utilize an auxiliary file, it might be desirable to include a field in the record format that indicates the variant of the key that was used to encrypt the data.

This works best with additional coding in the application programs to accommodate the possible existence of different key versions and the determination of the current version.

If translation is required, a conversion program can selectively read records encrypted under the *old* version of the key and systematically decrypt and re-encrypt the data using the *new* version of the key. The program would then store the updated ciphertext along with the key version information.

Even if the data is not translated in a single, dedicated pass by a conversion program, the application programs are still able to decrypt the data (regardless of which key version was used to encrypt it) and then re-encrypt using the latest key. This effectively adds the capability of translating the data upon *first touch*.

**Note:** The key *version* is not the actual key, but a reference for use by the encryption/decryption routine to determine which key was last used to encrypt the data. Storing the actual key in the database along with the ciphertext would enable a hacker to easily gain access to the plaintext data.

Figure 7-5 represents how an auxiliary file defined in DDS might contain key version information.

```

.....A*. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
***** Beginning of data *****
A*
A          R EMPXTR          TEXT('EMPLOYEE EXTENTION')
A          EMPID            10A      COLHDG('ID Number')
A          EMPKVER          15A      COLHDG('Key version')
A          EMPSSN           16A      COLHDG('ENCRYPTED SSN')
A          K EMPID          CCSID(65535)
A*

```

Figure 7-5 Incorporating a key version field into an auxiliary file

### 7.3 Converting the plaintext data to ciphertext

If it is necessary to modify fields or to normalize to auxiliary files, you are required to convert your data to match the new database layout.

Also, until every record has been processed by the encryption algorithm, the plaintext data is still visible and vulnerable.

This section describes the main tasks involved in accomplishing both tasks efficiently.

## 7.3.1 Adjusting to database structure changes

**Note:** If the database does not require any modification, you can skip this topic.

The first stage in converting to an encrypted state is to update the data and application to be compatible with the new database structure.

The process used to accomplish this depends on whether the changes are made to the existing record layout, or whether new auxiliary files are being utilized.

### Record layout modification

If all of the fields to be encrypted already had the desired data type, and you only needed to adjust field lengths, you may be able to use the Change Physical File (CHGPF) CL command or SQL's ALTER TABLE instruction to regenerate the file object and remap the existing data stored in the file.

This is accomplished by supplying the name of the updated DDS source member used to define the file.

Before using the CHGPF command or ALTER TABLE instruction, you should understand the functionality of each command. Important considerations include:

- ▶ It can be long running if the file has a large number of records.
- ▶ It requires an exclusive-no-read lock, which means no one can be using the file for any purpose.
- ▶ Data is mapped by field name. If you alter a field name, the data in that field will be lost during the conversion.

As such, we strongly recommend saving the file before issuing the command.

If the modification entailed data type changes, you need a conversion process to translate the data from the old data type to the new one. This can be performed via a program that you write, or via a copy process such as the one outlined below.

**Note:** The steps outlined below are an example of one approach to migrating data field lengths and data type.

Before you utilize any methodology, it is important to understand the ramifications of recreating database file objects. Considerations should be made about the impact that the size of the data members has on the migration time, as well as the availability of data during the migration process.

Not adequately testing the migration technique increases the likelihood that data will be lost or corrupted.

The steps are:

1. Back up the original file object.

This backup can be performed via saving the file object (and data) to tape, to a save file, or via a simple duplication of the file using the Copy File (CPYF) CL command or the Create Duplicate Object (CRTDUPOBJ) CL command.

This ensures that you have the ability to restore the file if necessary.

2. Document any unique characteristics to be reapplied after the file is recompiled. For example, you may want to review the following:
  - Database triggers
  - Referential constraints
  - Logical file information
  - Public and private authorities
3. Duplicate the original file source (DDS) member to a new and unique name.  
This intermediate file is used during the data *type* translation.
4. Modify the *intermediate* file for the correct:
  - Field length
  - Unpacked numeric data type (if currently defined as packed numeric)
 Do *not* change any numeric field data type to alpha/numeric at this point.
5. Compile the intermediate file.
6. Duplicate the records from the original file to the intermediate file using the Copy File (CPYF) CL command with the format option as follows:
 

```
CPYF FROMFILE(origflib/origfile) TOFILE(intmflib/intmedfile) FMTOPT>(*MAP)
(*DROP)
```

 This automatically maps the data by field name while accommodating discrepancies in field lengths and packed/signed numeric field data type.
7. Modify the original file DDS definition to have the same field lengths as the intermediate file. Now change any numeric fields that will store ciphertext to be alpha/numeric.
8. Add the following DDS keyword to any field that will store ciphertext:
 

```
CCSID(65535)
```
9. Recompile the original file using the DDS definition modified above.
10. Duplicate the data from the intermediate file to the new version of the original file using the CPYF command with the format option as follows:
 

```
CPYF FROMFILE(intmflib/intmedfile) TOFILE(origflib/origfile) FMTOPT(*NOCHK)
```

 The *\*NOCHK* option copies the data byte for byte, mapping any numeric information into the alpha/numeric field ready for encrypting.

Ensure that you also plan to:

- ▶ Recompile any logical views that reference the master file.
- ▶ Re-establish referential constraints and triggers and other custom file settings.
- ▶ Review public and private authorities to the new master file.
- ▶ Recompile all referencing objects (to prevent level-check warnings).
- ▶ Any other considerations determined during step 2.

## Database normalization

By moving the encrypted information to an auxiliary file, you avoid having to modify the original file layout and avoid the complications of recreating the original file object.

A conversion program is required to extract the plaintext data from the original field and relocate it to the auxiliary file. This is usually a simple read/write process. The original field can either be cleared of its sensitive data or repopulated with an index value to link each record to the associated record in the auxiliary file. This is required if the original file does not provide a unique key to its records.

If an application program references the modified fields, it must be reviewed to ensure that it correctly handles the alternate field type or field length. In addition, normalization requires the application programs to be modified to access the information in the auxiliary file, using the appropriate key or index information from the original file.

If, for some reason, the data extraction to the auxiliary file is not able to be accomplished in a single pass, you may wish to consider including functionality in the modification to accommodate the data retrieval from the original field, if the auxiliary record is not found.

### 7.3.2 Encrypting existing data

After any necessary remapping of plaintext data to a new field definition or to an auxiliary file, the next step is to encrypt the plaintext data into ciphertext.

For clarity, this process is being explained as separate from the database modification. However, it is often possible to combine the two processes.

Encryption of new/existing data can be accomplished using any one of the methodologies outlined below:

- ▶ Conversion program

If you wish to perform encryption as a conversion process, it can be summarized as a read-encrypt-update process over the file currently containing the plaintext data.

Significant (long running) conversions might need to be performed in phases, and this can be accomplished by enabling the application program to handle the existence of either plaintext or ciphertext data in the same field.

- ▶ First touch

By coding the application to read the plaintext data and then write/update as ciphertext, the data is gradually converted. The application needs to handle the existence of either plaintext and ciphertext in the same field until all of the data is encrypted.

Consideration also needs to be given for the time anticipated for all of the data records to be touched. Combining first touch with a subsequent conversion pass ensures that all records are processed in the necessary time frame.

- ▶ Database file trigger

An update (\*UPDATE) or new record (\*INSERT) trigger that intercepts and encrypts the plaintext data before storing it in the file record is similar to the first touch approach described above. The advantage is that the application does not need to be modified to perform the encryption when writing/updating the data to the file.

Alternatively, by writing a simple read/update routine, the trigger program can be invoked for all existing records similar to the conversion program methodology.

Triggers may work well when plaintext is stored in auxiliary files, as that type of file is only accessed when the encrypted information is required. However, if the encrypted field is in the same record layout as the rest of the data, that trigger is invoked for *every* database update, even if the record was not accessed for the encrypted information.

Any process that converts a large number of records en masse may be very time consuming and resource intensive. Processing only a test subset of records from a large file provides a useful estimate of the time required to encrypt all of the existing file data.

If the conversion effort requires more time than the conversion window allows, then you may employ a technique to reduce that conversion time.

### 7.3.3 Reducing the initial data conversion window

The time required for data conversion is dependent on many factors such as the size of the file, available processing resources, and the methodology used to convert the data. In many cases, the initial conversion (the act of taking all of the existing data that contains plaintext and replacing with ciphertext) can take a significant amount of time.

The most time-intensive component of the conversion effort can be attributed to the encryption algorithm as it generates the ciphertext. You may want to perform conversion benchmarking and employ some type of window-reduction technique such as the ones described below.

#### Phased conversion

When most programmers write code to accommodate ciphertext data, they are modifying the application in anticipation of *only* processing ciphertext. However, with some planning, the standard application modification to decrypt the ciphertext can be enhanced to perform the decryption only if the data is no longer being stored in plaintext form. That requires that the program be able to differentiate between the plaintext and the ciphertext, a topic covered in 8.2.2, “Determining encryption state” on page 100.

By enabling the program to handle both states of the data, the conversion can happen in multiple passes.

This approach is a way to reduce/eliminate the conversion process by performing the conversion of plaintext to ciphertext upon *first touch* of the application. This may only be viable if all of the data is accessed relatively frequently. If any information is historical in nature, then you need to consider how to convert those records.

#### Temporary mapping file

Depending on the type of information that is to be encrypted, one method likely to significantly reduce the conversion time window is to build a file of all of the plaintext and corresponding ciphertext values. By separating the encryption process from the mass database update, it is possible to perform the resource-intensive encryption process over an extended period, even while the database is in use.

Some suggestions are:

- ▶ Create a new library that is secured using normal i5/OS security mechanisms.
- ▶ Secure mapping files using normal i5/OS security mechanisms.
- ▶ Audit access to the library and mapping files and review those entries.
- ▶ Use adopted authority to enable access by the update programs during the conversion.
- ▶ Delete the mapping files upon completion of the final conversion.

**Note:** If you use this approach, you must be cognizant of the vulnerability of having files of this nature on your system. It is critical that the file be secured by object-level security during the conversion process, and then deleted upon completion of the conversion.

Just as importantly, the mapping file must be excluded from any saves. Gaining access to this data would provide significant insight into your encryption process by facilitating a *known plaintext* type attack, as well as enabling the simple retrieval of the plaintext data.

An example of a simple mapping file that might be used during the encryption of a Social Security number is shown in Figure 7-6.

```

.....A*. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
***** Beginning of data *****
A*
A      R EMPSSNXR          TEXT('EMPLOYEE SSN XREF')
A*
A      SSNPLAIN          9A      COLHDG('Plaintext')
A      TEXT('Plaintext')
A      SSNCIPHER          16A     COLHDG('Ciphertext')
A      TEXT('Ciphertext')
A      CCSID(65535)
A      K SSNPLAIN
A*

```

Figure 7-6 Social Security number mapping file

The mapping file methodology requires you to create an extract process and an update process, as represented in Figure 7-7.

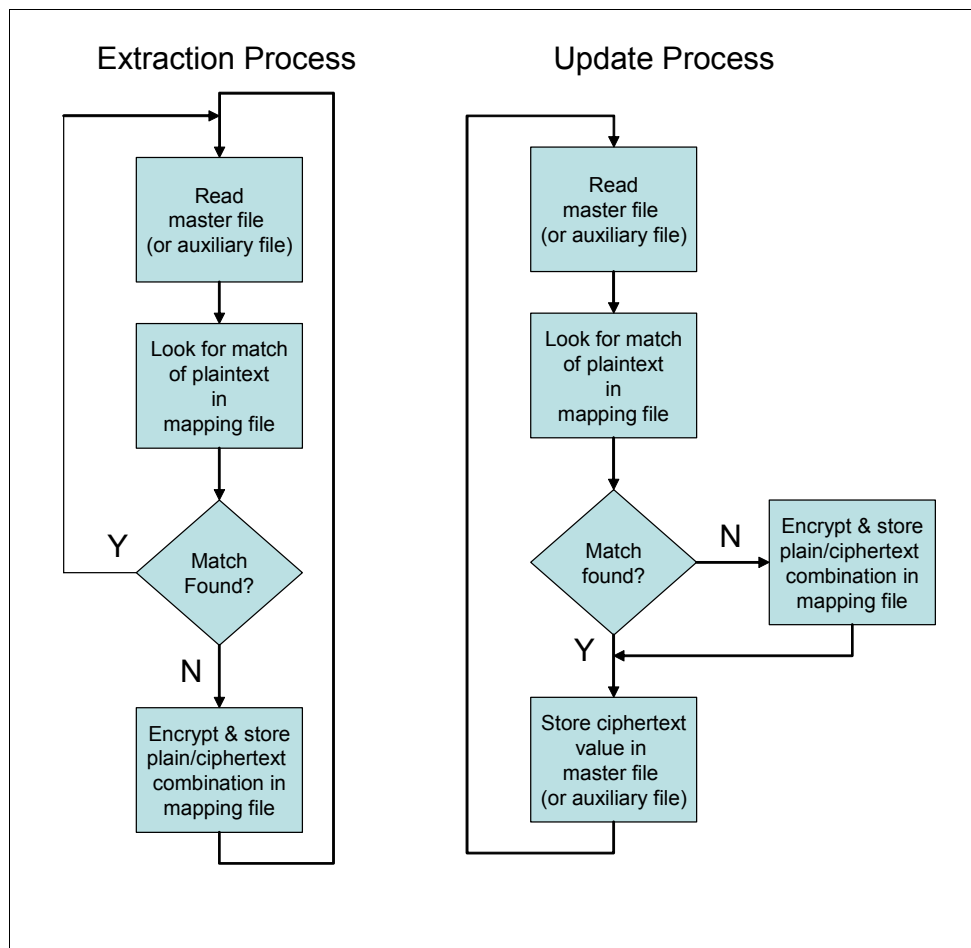


Figure 7-7 Using a temporary mapping file to reduce the initial conversion window

The conversion is broken down into two main phases, extraction and update, as explained below.

► Extraction process

The extract process reads through the master (or auxiliary) file for all variants of the plaintext field values to be encrypted.

The program should check for the existence of the plaintext value in the mapping file, and any plaintext value not found should be written along with the associated ciphertext.

The existence check enables the extraction process to:

- Fill in any missing plaintext values that have been added to the original/auxiliary file between multiple extraction runs.
- Prevent the storage of duplicates of the plaintext value.
- Process very large data files by breaking the extraction down into multiple runs.
- Be restarted after a critical error.

► Update process

After the mapping file is built, a simple record update process is required to update the master (or auxiliary) record to replace the plaintext with the applicable ciphertext.

This program is extremely efficient, as it only has to perform fast random-access record retrievals using the plaintext as the key to locate and store the associated ciphertext value.

We recommend that the update program include processing to handle any plaintext values that are not located in the mapping file, possibly caused by values added since the last extract run. Exception handling may include reporting the exceptions or performing an *on-the-fly* encryption and write of the missing information to the mapping file.

### 7.3.4 Validating the encrypted data

If you chose to use a mapping file, then a validation program (or an additional routine in your extract program) can easily be used to validate the encryption, as well as decryption, processes.

By reading the mapping file and decrypting the ciphertext back to its plaintext value, it is possible to validate that the original encryption (and subsequent decryption) was performed correctly.

Enhancing the mapping file to store the decrypted value allows a query-type match to be performed against the original plaintext value. If either the encryption or decryption processes do not work correctly, then the original plaintext and decrypted (validation) value will not match.

```

.....A*. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
***** Beginning of data *****
A*
A          R EMPSSNXR          TEXT('EMPLOYEE SSN XREF')
A*
A          SSNPLAIN          9A          COLHDG('Plaintext')
A          TEXT('Plaintext')
A          SSNCIPHER          16A         COLHDG('Ciphertext')
A          TEXT('Ciphertext')
A          CCSID(65535)
A          SSNDECRYPT          9A          COLHDG('Validation')
A          TEXT('Validation')
A          K SSNPLAIN
A*

```

Figure 7-8 Social Security number mapping file enhanced for validation

If no discrepancies are discovered between the two plaintext fields, then this file can now be used safely when performing the data conversion.

If you do not utilize a mapping file, then you must test the encryption/decryption process across representative *test* data to ensure that the ciphertext can be returned to the original plaintext value. The means by which you verify depend on the methodology used to process the data.

The challenge of trying to validate the encryption and decryption process when not using a mapping file is that the ciphertext usually replaces the plaintext value. A possible workaround might include matching decrypted values against a backup copy of the original file with plaintext data. Another option would be to store a hash of the plaintext data. When the ciphertext is decrypted, a hash can be calculated and compared with the stored value.

## 7.4 Common tools for data maintenance and inquiry

Although the vast majority of data manipulation and inquiry is performed by application programs designed specifically for that task, a number of generic utilities exist to allow a user with the appropriate i5/OS authority to have direct access to the data within the file.

These utilities are frequently impacted by the inclusion of encrypted data within a file due to the expanded character set utilized when the data is in encrypted form. Some characters preclude the display and editing of the file data via a display window.

This section reviews some of the most common tools used to access file data, along with considerations to allow their continued use.

### 7.4.1 AS/400 Data File Utility (DFU)

DFU is a program generator that allows you to create programs to add new records, update existing records, and delete records from a file. It also allows you to inquire about the data stored in the file, though there are tools that provide far greater capability in this area.



A DFU program can be generated without requiring any knowledge of a programming language by you supplying prompted information about what capabilities the DFU program should have.

DFU also provides the ability to manipulate data in a file without having to first create a DFU program. In this case, a temporary program is generated and run by DFU automatically using default settings.

**Note:** Unlike modifying plaintext data (which can often be corrected), modifying any element of the ciphertext usually renders part of the data meaningless, as it no longer decrypts to the original value. Even if the change is detected, correcting it can often be impossible, as many characters used in the ciphertext are not accessible via the keyboard.

For more information about the use of DFU programs, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/>

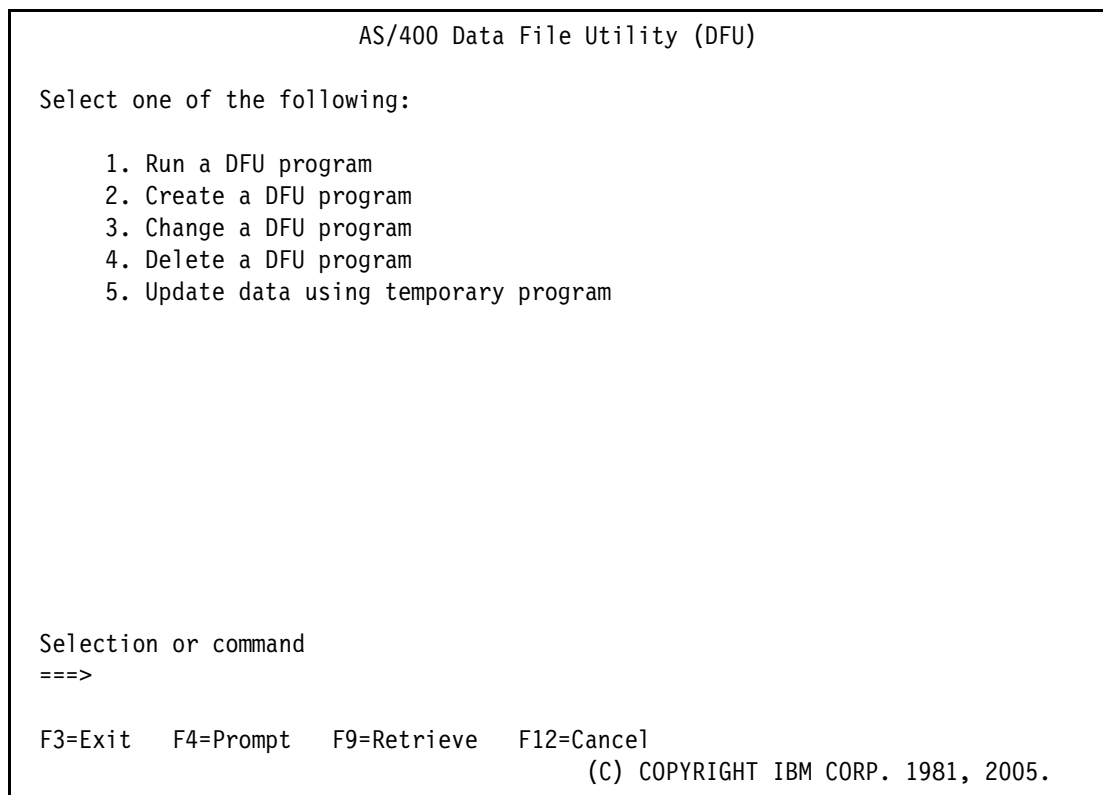


Figure 7-9 AS/400 Data File Utility main menu

### Temporary DFU programs

Temporary DFU programs allow a user to access the data in a file without having to provide any information beyond the name and location of the file. This is a popular approach when the requirement to access a file is not anticipated and is not expected to be repeated. However, allowing DFU to create temporary programs over files that contain encrypted fields will result in decimal data errors and the appearance of the following error message:

The retrieved record contains invalid data.

The error is caused by the inability of the 5250 data stream to correctly represent the binary encryption data. The encrypted data may even include escape sequences that prevent subsequent fields in the record format from being displayed.

If you wish to continue to utilize DFU to provide editing capabilities over files that contain encrypted fields, then you have two options:

- ▶ Create a logical file (LF) view over the file that omits the encrypted fields.
- ▶ Utilize a *permanent* DFU program to edit the file.

### Permanent DFU programs

Permanent DFU programs may be able to present encrypted data for display, based on their ability to suppress (decimal data) errors, as shown in Figure 7-10. We say that it *may* present the data as, like temporary DFUs, it is possible that the ciphertext will contain escape sequences that will cause unpredictable results.

When DFU is instructed to suppress errors, an algorithm is used on invalid data in an attempt to correct it. The *corrected* data is displayed on the screen and no error message is displayed. The data that is displayed may not necessarily be the data in the field. However, changes are only recorded to the database if you change the field's data yourself.

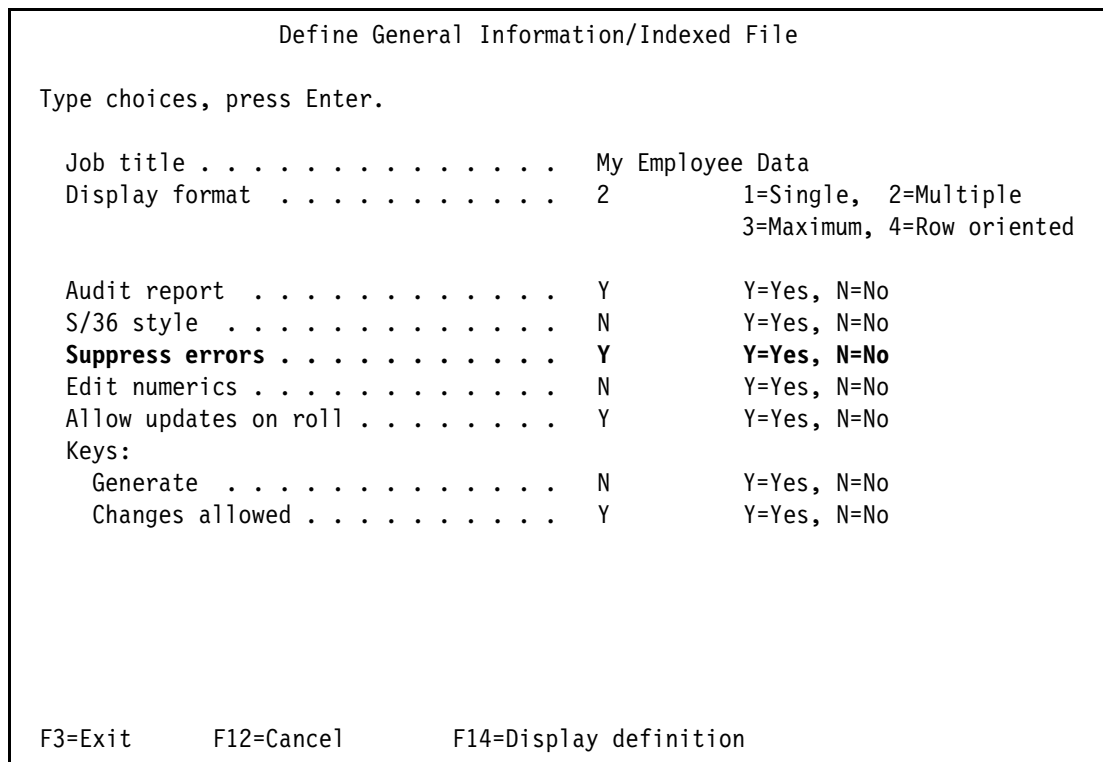


Figure 7-10 Directing a DFU program to suppress errors

However, in light of the fact that modifying encrypted data outside of the intended application can have serious side effects (including losing the ability to decrypt the data), we strongly recommend reviewing whether the inclusion of the encrypted data is necessary and appropriate.

Permanent DFU programs permit the selective display of fields from the record format, as well as the ability to render a field as read-only. We recommend that DFU programs be

configured to suppress encrypted fields from the display entirely, to remove the risk associated with a user changing (and thereby corrupting) the ciphertext.

If there is a concern that a DFU user might delete a record, where the only data is stored in a field that is encrypted but suppressed from the screen, presenting the field in a read-only mode might be an appropriate compromise.

## 7.4.2 IBM Query for i5/OS (Query/400)

Query is an inquiry and reporting tool designed to allow rapid development of reports and data extractions. It has some distinct benefits that include:

- ▶ It is easy to use.
- ▶ No programming experience is required.
- ▶ Observation of i5/OS object-level security.
- ▶ Queries can be stored and executed multiple times.
- ▶ Queries can be scheduled to run at specific dates/times.

However, it also has some significant security considerations, including:

- ▶ Permits access to data outside of application control.
- ▶ Reliant on correctly implemented i5/OS object-level security.
- ▶ The ability to route query results to an external file presents integrity risk if the user routes the exported data back into the original master file.

The risks grow exponentially if a file is not correctly secured by i5/OS object-level security.

For queries that are designed to present their results to the screen or to a spooled file, encrypted fields must be removed from the query definition. This is not normally a problem, as encrypted data is usually meaningless in both mediums.

However, if the design requirement is to present the data in plaintext form, then you can use an SQL view. If you do not wish to use SQL, you must first decrypt the file data and *then* query it. If your query definition performs record selections or file joins, and the subset of data is likely to be significantly smaller than the master file, it may be faster to first extract the selected records into an intermediate file, and then decrypt the results. After that, you can execute a secondary query over the intermediary file to generate the required report.

You need to be aware of the additional time that this will take to perform, as well as the requirement to have a highly secure program control the execution if you choose to leverage multiple queries and a decryption routine.

Also, the use of an intermediate file containing the plaintext data poses a security exposure for as long as the data exists in a plaintext state. If an intermediate file is absolutely necessary, we recommend that it be created in the QTEMP library, although, if you are running your system at security level 40 or lower, then it is still possible for this area to be accessed by other users or for it to be orphaned if the job ends abnormally. An IPL or a Reclaim Storage (RCLSTG) would perform the necessary clean up. At security level 50, the RCLSTG is required as, at that level, the system does not maintain the pointers to the storage location that are needed by the IPL, to locate and clean up the information

For queries designed to extract data to another database file, the benefit of including the encrypted data is dependant on the purpose of the extraction. Again, be aware of the integrity vulnerability associated with performing queries to output files.

You will need to change any query definitions that performed joins via the data that is now encrypted, to use another data element for the join.

Generally, the safest solution for a query definition that needs to print or display encrypted data in its plaintext state is to replace it with a program to perform the decryption and output processing.

### 7.4.3 Interactive SQL

Working with data via the Structured Query Language (SQL) interface is a common and practical approach, especially when the file contains a large number of data records.

Think of interactive SQL as the more functional brother of Query/400. Although SQL is more of a language-driven tool compared to Query/400's user-friendly interface, it provides many of the same functions. However, the power of SQL comes with its additional data manipulation abilities. Unlike query's confinement to reading data, SQL has the ability to insert new records, as well as update and delete existing ones. In fact, SQL can be used to define the database itself.

Similar to query, when the native SQL interface encounters a column containing encrypted data, it is often not able to display the column, or even the entire record, due to possible includes of field codes in the binary data that comprises the ciphertext.

If the inclusion of encrypted columns is not a requirement of the SQL operation, selecting only the required (plaintext) columns allows the data to be displayed without issue. As an example, general selection statements such as the following should be modified to only select the plaintext columns:

```
select * from EMPPF
```

It might become:

```
select empid, emplname, empfname, empdept from EMPPF
```

If the SQL statement requires the inclusion of the encrypted data in plaintext form, it is possible to use a built-in function to perform the decryption as part of the SQL operation.

**Note:** For more information about using SQL to perform cryptographic functions, refer to Chapter 10, "SQL method" on page 115.

### 7.4.4 Other tools

Other popular tools for data display and manipulation are:

► RUNQRY

As well as providing the ability to execute stored Query/400 definitions, the Run Query (RUNQRY) CL command provides a way to access file data in a formatted form without having to first create a query definition by supplying a query definition name of \*NONE, along with the desired files name and location.

As with Query/400 definitions that include encrypted fields, this command will have unpredictable results when encrypted data is encountered.

► RUNSQLSTM

The Run SQL Statement (RUNSQLSTM) CL command processes a source file containing any number of SQL statements.

The restrictions for using this command with encrypted data are the same as for interactive SQL.

► **DSPPFM**

The Display Physical File Member (DSPPFM) CL command is the standard mechanism for displaying unformatted data in a file from a 5250 display.

This command will display data in a file that includes encrypted data. The encrypted data will be represented in a non-readable form, but all other fields will be displayed correctly.

By pressing F10 during the display of ciphertext, you will be able to view the HEX values for the data.

► **iSeries Navigator**

The iSeries Navigator graphical interface provides rich functionality related to database design and maintenance. The view and edit function available under the databases category will present the ciphertext via its HEX values. This allows the record to be made available for viewing and editing, although we do not recommend direct editing of the ciphertext.

In general, existing file utilities can continue to be used, as long as they provide the capability to select which fields to present to the display or display the ciphertext in its hexadecimal form.





# Application considerations

This chapter discusses the programming considerations for earlier applications when working with data stored in encrypted form.

## 8.1 Accommodating database changes

As you (re)design your database structure to be compatible with the storage of ciphertext, any changes that you make will have an impact on the programs that reference it. As we mentioned previously, it is necessary to have a documented overview of where the affected files are used and how they are used. The changes required will depend on whether the existing database file was modified or whether the ciphertext is to be migrated to an auxiliary file.

### 8.1.1 Record format changes

If you choose to modify the existing record format, the initial impact on the referencing programs will be dependent on how the file is defined to the program.

If the application uses external definitions, then the references will be updated when the program is next compiled.

If the file is defined internally, then the program will either:

- ▶ Reference the ordered fields by length.

For field references, the starting and ending positions are inferred from the order of the fields in the definition and the length and data type of each field.

By updating any altered field lengths and data types, the positions for all fields in the record format will be recalculated during the compile phase.

- ▶ Reference by the starting/ending positions in the record format.

If the fields are defined by the starting/ending position of the current fields, then you must redefine the new positions of the modified fields along with all fields that follow in the record format.

### 8.1.2 Database normalization

By using an auxiliary file to store the ciphertext version of the data, you will not have to modify any referencing programs for changes in the file definitions of the *main* file.

If a program does not require access to the encrypted data, then the program will not be required to access the supplemental record stored in the auxiliary file. When assessing use of the encrypted data, do not overlook whether data is transferred from one file to another by using matching field names in both files.

If a program requires access to the encrypted data (in either ciphertext or plaintext state), you will need to define the auxiliary file to the program and code the retrieval of the auxiliary record based on the unique file key. If it is not possible to uniquely identify a record in the original file, then the field whose data was externalized should be considered for use as an index to the associated record in the auxiliary file.

## 8.2 Working with encrypted data

When working with data in an encrypted form, there are likely to be a variety of changes to the application function and flow. The magnitude of the changes is dependent on how the application uses the encrypted data.



This section reviews some areas of design that need to be reviewed in most encryption projects as well as some possible ways to work through them.

## 8.2.1 Performing encryption tasks with database triggers

A database (physical file) trigger is a mechanism that invokes a program when a specific action is performed on a database record.

There are four event actions that can be reacted to as well as two potential timing selections of each. Table 8-1 outlines the valid combinations of events and timings.

Table 8-1 Valid combination of external trigger events and timings.

Event/timing	*BEFORE	*AFTER
*READ	NO	YES
*INSERT	YES	YES
*CHANGE	YES	YES
*DELETE	YES	YES

Triggers can be defined in two different ways:

- ▶ SQL triggers
- ▶ External triggers

**Note:** For practical information about creating and working with both types of triggers, refer to the Redbooks publication *Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries*, SG24-6503.

It is tempting to consider the use of a trigger to perform *all* of the encryption/decryption functionality, removing the requirement for any application modification. Although triggers might be the *only* solution available if you do not have access to your application source code, it is not always a *silver bullet* solution.

One major issue is the performance implications associated with using a trigger. Depending on the number and type of I/Os performed on the file, a trigger can add significant overhead to the application. This is due to the fact that a trigger is invoked for *every* operation related to the trigger event, even if the data is not being accessed for the encrypted information.

External triggers are created using the Add Physical File Trigger (ADDPFTRG) CL command. SQL triggers are created with a CREATE TRIGGER statement.

### Using an external trigger to encrypt

Creating an \*INSERT and \*CHANGE trigger on a file in \*BEFORE mode permits the program associated with the trigger to take the plaintext data and convert it to ciphertext before the database engine writes the data to the file.

A trigger designed to encrypt data is one way to perform the function without requiring modification to source code, as the encryption operation is transparent to the application program.

If the database requires any structure modification, then any referencing application program is also required to be recompiled, or the file needs to be set to ignore level checks. The

decision to ignore level checks generally requires an understanding of the application that is not attainable without having access to the application source code.

Alternatively, a BEFORE trigger that is invoked by an INSERT or CHANGE event can intercept a request to store private information in plaintext, and invoke a program that might perform the following tasks:

- ▶ Perform the encryption.
- ▶ Store the ciphertext data to an auxiliary file.
- ▶ Clear the original plaintext field from the original write/update request (or use it to store an index to tie it to the auxiliary record).

If the database was already compatible with the format required for the ciphertext, then the trigger can merely perform the encryption and replace the plaintext with the ciphertext in the AFTER image buffer, and then allow the database manager to write the data to the file.

### Using a trigger to decrypt

A popular feature of a trigger is that it invokes the trigger program regardless of the mechanism that was used to process the data: application program, SQL, DFU, and so on. This is because the trigger effectively intercepts the request inside the i5/OS database engine.

You cannot simply use an external trigger event to return the data in plaintext, as a READ trigger does not permit the data buffer to be altered by the trigger program. This is by design and, at first glance, may seem like a limitation. However, if you stop to consider that a rogue program could completely misrepresent the actual file data to *every* mechanism that accesses that data, it is a legitimate one.

If you must use a trigger to decrypt data, you can use an SQL view to perform the decryption in a user-defined function (UDF). For more information about using SQL to perform cryptographic functions, refer to Chapter 10, “SQL method” on page 115.

## 8.2.2 Determining encryption state

Field data can exist in one of two states, encrypted (ciphertext) or decrypted (plaintext). One of the issues that this presents to a programmer is knowing which one they are dealing with. It seems obvious when we visually inspect the data, but we may also need to be able to determine the data state programmatically.

An example is when a field moves through a program by jumping in and out of a series of work fields, in various states of encryption, before being written out to a secondary database file. Before writing the record, it might be beneficial to validate that the data is in the correct state. This is especially true in monolithic applications where the complete path of the field value might be extremely hard to follow.

Realize, however, that determining the encryption state is more of an art than a science, and your success depends on the complexity and integrity of the data in your database.

### Character analysis

Data that is encrypted usually has a much broader character set than its plaintext equivalent and may include characters that are not available via the keyboard. One simplistic way to detect the encryption state is by analyzing the characters included in the data. For example, a Social Security number in the United States is a 9-digit number containing characters in the

range 0–9. If the program performs a data validity check and determines that there are other characters present, then it may be assumed that the data is in an encrypted state.

**Note:** This is not a foolproof way of determining encryption state, as it is reliant on the integrity of the plaintext data.

Remember to include any special characters that might be used to format the data. For example, a telephone number usually comprises the digits 0–9, but, depending on your database, may also need to accommodate blanks (leading, trailing, or embedded), as well as other common formatting characters such as '() - '. Omitting these from the validation string might falsely identify the data as being in an encrypted state.

Figure 8-1 shows a simple RPG example of performing this character analysis on a Social Security number field (SSN).

```
D*
D* Set up program variables
D*
D*   Valid_Chars - Characters valid for plaintext
D*   SSN         Social Security Number
D*   State       Encryption state
D*
D Valid_Chars    C                CONST('0123456789')
D SSN            S                9A
D State          S                9A
C*
C* Receive the SSN from a program parameter
C*
C   *Entry      Plist
C               Parm              SSN
C*
C* Set the default state to be plaintext and then evaluate the
C* field to determine if the data has characters outside the valid
C* data range suggesting encryption
C*
C               Eval      State = 'Decrypted'
C*
C               If        %Check(Valid_Chars:SSN) > 0
C               Eval      State = 'Encrypted'
C               EndIf
C*
C* Display the field state
C*
C   State       DSPLY
C*
C               Eval      *INLR = *On
C               Return
C*
```

Figure 8-1 Simple RPG program sample to determine encryption state of a field

If this type of character analysis needs to be performed numerous times in the code, it might be beneficial to incorporate the logic into an ILE procedure to simplify the mainline code. Combining this with other procedures (such as a decrypt function) might resemble the RPG example in Figure 8-2.

```
C*  
C* Determine the plaintext version of the data  
C*  
C           If           encryption_state(SSN) = 'Encrypted'  
C           Eval        plaintext = decrypt(SSN)  
C           Else  
C           Eval        plaintext = SSN  
C           EndIf  
C*  
C*
```

Figure 8-2 Using ILE functions to perform encryption-related tasks

If a program accesses multiple sources of encrypted data, then the application could designate separate procedures for each one, so that the validation can be appropriate for the particular field. This approach also permits the encryption routines to be modularized and secured appropriately using i5/OS security.

### Field data tagging

Another approach is to utilize an unusual (fixed and unique) string of text and append it to the actual (variable) data. This enables the programmer to determine the encryption state by programmatically analyzing the bytes assigned to that position of the data to see if it matches the value known to the program. For example:

1. The application user enters the telephone number 515-555-8838 at the workstation.
2. The program appends the unique string '%TEL%' to the unedited data. The resulting string is 5155558838%TEL%.
3. The program encrypts the result and then stores the ciphertext value in the database.
4. The program later retrieves the data. At any point in the application process, regardless of the number of field movements, the program can interrogate the last 4 bytes of data. If the value is '%TEL%' then the assumption is that the data is currently plaintext.

When using this approach, you must factor in the additional bytes required for both the plaintext and cipher text data in the program functions as well as the database file.

Also, any actual uses of the data need to have the tag string stripped off prior to display/printing, and so on. Removing the tag string is easier if it is placed in a fixed position. If used as a suffix, then this may mean blanks or zeros appearing between the actual data and the tag string.

### Field data length

When data is in an encrypted state, it is likely to be longer than the plaintext equivalent. By determining the length of the data in the field, we can get an indication of the encryption state.

**Note:** This is not foolproof, as, depending on the selected encryption algorithm and plaintext field length, it is possible that the ciphertext length will equal the plaintext length.

### Parity tagging

One way to increase the reliability of the field data length approach is to combine it with data tagging. This might include the calculation of the length of the plaintext data and then appending that information as the tag.

When the program needs to interrogate the encryption state, it checks the length of the data (excluding the tag) and then compares that value to the tag. If the two values match, then the data is most likely plaintext.

The approach has the advantage over the field data tagging technique by not requiring a fixed (and known) value to be placed in the string and encrypted. Having known text in a ciphertext field increases the possibility of the encryption being broken.

### 8.2.3 Data sorting

If your application performs processing or utilizes displays that are dependent on the data being presented in order of the fields' plaintext value, then this will no longer function as expected.

The challenge for the application is that, even if the ciphertext is decrypted back to the plaintext state before being written to the screen or printed on a report, the sort order of the records will appear to be completely random. For example, Table 8-2 shows five names stored in a database file (Jordan, Sydney, Bethany, Leeann, and Brandon) along a simple ciphertext equivalent. The file is keyed by the name field.

Table 8-2 Sample plaintext to ciphertext for example

Plaintext	Ciphertext
Jordan	KptFSm
Sydney	DufMrU
Bethany	nRYjSmU
Leeann	KRtSMm
Brandon	NtSmFPm

When this data is sorted in its plaintext state, it is read in the order shown in Table 8-3.

Table 8-3 Data sorted by plaintext value

Sort field
Bethany
Brandon
Jordan
Leeann
Sydney

However, when this data is converted and sorted in a ciphertext state, it is read in the order shown in Table 8-4.

Table 8-4 Data sorted by ciphertext value

Sort field	Which decrypts to
DufMrU	Sydney
KptFSm	Jordan
KRtSMm	Leeann
nRYjSmU	Bethany
NtSmFPm	Brandon

As you can see, once the plaintext state is retrieved, there is no alphabetic sequence to the data.

### Subfile-based applications

A subfile is a 5250 display mechanism that presents application data in a form that allows the user to *page* through a number of records using the roll up/down or page up/down keys on the keyboard.

There are two main types of subfile-build methods used in an application:

- ▶ Build once

This type of subfile is built by an application as it reads all of the available data in a single pass. If the maximum number of records is a known entity, then this is the simplest method to program for. For performance reasons, *build once* subfiles are traditionally reserved for processing a data set containing a relatively small number of records. Once the subfile is presented to the display, the screen manager controls the page up/down requests and control is only returned to the application program when the application user requests an option or presses a function key. This subfile can contain a maximum of 9,999 records.

- ▶ Page-by-page

Using this method, the application program reads only enough records to fill one page (or screen) on the display. By working on the premise that the average application user will often only page down once or twice to access the next page of records, they are fast and efficient.

There are two subtypes of page-by-page build:

- Page < size

This type of subfile adds speed to the simplicity of the build once approach. Now, the display manager returns control to the application program, when and if the user attempts to page *down* past the last page of records, to add one or more additional pages of records to the subfile. Like the build once method described above, this subfile can contain a maximum of 9,999 records, as new records are appended to the end of the existing subfile.

- Page = size

This subfile only ever contains one page of records. When the user pages *in either direction*, the display manager passes control to the program to rebuild the subfile with the desired page of records. Although the most resource-efficient method, this is often the most complex to write and is therefore not as common as the other techniques.

The main advantage of this method is that there is no maximum to the number of records that can be handled.

## Correcting data sequence

There are several methodologies that can be employed to ensure that the data is represented in the correct sort sequence:

- ▶ Sort on an alternate field.

One solution is for you to modify the application to present the data sorted on an alternate (non-encrypted) field. This keeps you from having to sort the plaintext information *after* it is decrypted.

- ▶ Sort APIs.

i5/OS provides the QLGSORT and QLGSORTIO APIs to sort data in a file or in a memory buffer. Using normal data access techniques, you modify the application to read and decrypt the required data. After that, you can utilize the APIs to resequence the data prior to display or printing.

If you have a subfile application that performs a page-by-page build, then you should modify it to be a build once subfile. If not, only the current data in the subfile will be sorted, and that might only represent a small fraction of the entire data set.

If this modification causes the total number of data records to exceed the capability of a subfile (9,999 records) or has unacceptable performance implications, then you might wish to change the program to perform a preliminary data selection (using a non-encrypted field) to reduce the number of records to be included in the subfile. For example, ask the application user to enter a postal code before listing account names.

For more information about using the QLGSORT and QLGSORTIO APIs, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/topic/apis/nls2.htm>

For a practical example, using the sort APIs for a subfile sort, refer to the Code400.com Web site:

<http://www.code400.com/qlgsort.php>

- ▶ SQL view.

You can modify your application to replace the native I/Os with SQL-based I/Os. Using a SQL view, your application can access the pre-sorted plaintext data.

For more information about using SQL to perform decryption tasks, refer to Chapter 10, “SQL method” on page 115.

- ▶ Intermediate file.

If your application has to read through a very large number of data records (for example, greater than 9,999 subfile records) in a valid plaintext sequence, and you do not have a way to subset that data, then an additional option is to decrypt the data to an intermediate file and then read that copy instead of the original.

This has obvious performance and time considerations, as the entire file has to be read and decrypted before the program can even start its original function.

It also presents a security vulnerability, as the data exists on disk in decrypted form. If you choose this approach, place the intermediate file in library QTEMP during the job's execution to help reduce the exposure.

## 8.2.4 Random access to encrypted data

Applications that need to perform keyed access to files that are keyed by fields that are now encrypted have similar challenges to programs that sort on the data in encrypted fields.

Again, this is due to the plaintext and ciphertext representations not having any correlation to each other.

To resolve this, the application must be modified to retrieve the data using another field as the key. Or encrypt the plaintext data and search using the ciphertext.

## 8.2.5 Triggers

When triggers are defined for a database file, *every* mechanism or application that can perform the function monitored by the trigger will invoke the trigger application.

If your application relies on database triggers to perform database functions, then the presence of encrypted data should not significantly impact the function of the trigger programs.

You must ensure that the trigger program does not attempt to process or change the ciphertext, or it may render the data undecipherable.

Triggers used for performing cryptographic functions are discussed further in “Using an external trigger to encrypt” on page 99.

## 8.3 Other considerations

In this section we discuss other areas that should be considered with applications that utilize encrypted data.

### 8.3.1 Spooled files

Although not generally considered part of an application, a spooled file containing plaintext data is a medium that may represent a significant vulnerability in your organization.

Unfortunately, basic spooled file security is often overlooked by System i administrators. A lack of understanding of the Spool Control (\*SPLCTL) and Job Control (\*JOBCTL) special authorities mean that it is often possible for users to access private information outside of any protection afforded by the database. Even legitimate users are often given these special authorities when they require only a subset of their capabilities (for example, starting a printer writer), that could be performed easily via a fixed-function program.

Regardless of whether a spooled report is printed or retained electronically, it has one main purpose: to present information in human-readable form. Therefore, the only state that a programmer is ever likely to include on a report is plaintext. For this reason, we recommend that application programs coded to print private data in its plaintext state be engineered to either:

- ▶ Remove the data.

Try to justify removing private information from the report, as this is the only guaranteed way to eliminate the risk of accidental disclosure.

- ▶ Mask the data.



Masking the data reduces the vulnerability of private data that is required to be included in plaintext. For example, this may include modifying a customer invoicing application to present only the last four digits of a consumers credit card number (a feature permitted under the PCI standards).

For more information about spooled file and output queue security, refer to Chapter 6 of the *iSeries Security Reference Version 5*, SC41-5302, or the security category of the i5/OS Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iseres/v5r4/topic/rzahg/rzahgicsecurity.htm>

### 8.3.2 Exported data

Many applications are designed to export data out of the database and store the results in another file, even one that may reside on another server.

Although legitimate data exports provide additional value to your corporate data, it is important to ensure that these exports only include private data in plaintext state when *absolutely* necessary. An example might include sending data to a bank or government agency.

If the application generates a plaintext export of the data, ensure that intermediate files are removed immediately upon completion of the export process.

Sending an export file as a simple e-mail attachment over an unsecured connection or leaving the previous 12 months' worth of bank transfer files sitting in a decrypted state in a folder on the Integrated File System completely undermines any encryption of the physical file data.





## Backup considerations

When working with encrypted data, you must plan to ensure that backup data remains available and usable. First, you need to make encryption keys available for decrypting backup data stored in an encrypted form. Second, care must be taken to make sure backup data does not introduce vulnerability in overall security.

## 9.1 Managing keys on a backup system

In order for encrypted backup data to be usable, you must make decrypting keys accessible on the backup system. Keystores or validation lists used to access keys must be made available on the backup system. See Chapter 5, “Managing keys on System i” on page 47, for information about ways to move or distribute key data.

If the encryption solution makes use of encryption-card hardware, hardware must also exist on the backup system with the required keys in its store. Refer to “Moving a keystore file” on page 54 for information about how to move or distribute key data in this instance.

Your backup planning should also include a contingency plan for recovering keys that are compromised, lost, or accidentally destroyed. The encrypted data is effectively destroyed if access to the decrypting keys is no longer possible. Refer to 3.6, “Backing up keys” on page 32, for recommendations on saving key data.

### 9.1.1 Coordinating keys between multiple systems

Distribution of new keys and changes to existing keys must be coordinated when data is distributed among multiple systems. Refer to 3.7, “Changing keys” on page 33, and 3.8, “Key distribution” on page 34, for additional information.

When keys change, before the former key is destroyed, backup data must be decrypted with the former key and encrypted with the new key in order to retain access to the data.

### 9.1.2 Translating keystores

If a hierarchical key structure is used where keys are encrypted by other keys (see KEK), key translation resulting from changes to keys must also be coordinated on the backup systems. Refer to 3.7, “Changing keys” on page 33, for additional detail.

### 9.1.3 Transporting keys between systems

The most secure policy is not to transport keys between systems at all. In this case, keys are stored locally on each system and changes to keys are coordinated by the system administrators.

Avoiding transmission of keys is not always possible. If key data must be transported between systems, a good policy is to encrypt the key data using a KEK. Refer to 3.8, “Key distribution” on page 34, for additional detail.

## 9.2 Securing backup data

Data stored for backup purposes should not introduce vulnerability in overall security.

### 9.2.1 Transporting data to the backup system

Encrypted data stored to media such as disk or tape should be transported to the backup system separately from the key data needed to decrypt it. Storing the keys and encrypted data on the same media introduces a security vulnerability. If key data is stored on the same

backup media as the encryption data, the key data should be encrypted with another key not on the media using a KEK.

The authority granted for save files or files replicated for backup is an important aspect of backup data security. Your backup and recovery strategy should limit access to backup data to authorized users only.

Sensitive data transported to a backup system by a remote database can be protected using SSL encryption over the DDM service.

For encrypted data stored on an iASP device that can be switched between systems, the required keys must be made available on all systems in the domain of the iASP.

## **9.2.2 Working with encrypted data between multiple systems**

Data compression and data translation cannot be performed on encrypted data. For example, if the backup system has a different CCSID or different date and time format than the primary system, the data must be translated in its decrypted form. If the data is in an encrypted form, the data must first be decrypted before compression or translation is performed, then encrypted again.

Perform regular practice tests of restoring your data from backup in order to prove the effectiveness of your backup policies.





## Part 3

# Implementation of data encryption

This part provides various implementation scenarios with step-by-step instructions.







## SQL method

SQL provides for encrypting and decrypting data in a simple and convenient manner. However, SQL only provides a portion of encryption services. For example, functions that assist with the management of encryption keys are only available using native (non-SQL) interfaces. Even so, data can be made secure by using the capabilities provided in SQL only.

## 10.1 Preparing for encryption

Securing data using encryption introduces new requirements with respect to databases and applications. Changes to database formats are necessary to accommodate encrypted data. For a general discussion of database considerations, refer to Chapter 7, “Database considerations” on page 77. Applications require access to passwords or keys in order to access encrypted data. For a general discussion of application considerations, refer to Chapter 8, “Application considerations” on page 97. Here we consider the requirements of data encryption from the SQL perspective.

### 10.1.1 Encryption prerequisites

The first step is to identify exactly what data is sensitive and must be encrypted. Then we determine how the password or keys are made available in order to access the encrypted data. For example, assume that we have an employee table containing records of employee information:

```
CREATE TABLE emp (  
    employeeId CHAR(7),  
    ssn CHAR(9)  
    name VARCHAR(40),  
    salary NUMERIC(13)  
)
```

We decide to encrypt only the ssn and salary fields using the triple DES encryption algorithm. For simplicity, we decide to use a single password to encrypt these fields in all records of the emp table.

### 10.1.2 Identifying changes to your database

Encrypting data stored in a column (or field) within a DB2 table (or physical file) changes the definition of the column (or field). Data resulting from encryption is a binary string and requires the column to be defined as one of the following data types:

- ▶ BINARY
- ▶ VARBINARY
- ▶ CHAR FOR BIT DATA
- ▶ VARCHAR FOR BIT DATA
- ▶ BLOB

The CCSID value 65535 is used for encrypted data.

The column length (or field length) also changes as a result of encryption.

When using SQL to perform encryption, the length of an existing column increases because additional bytes are needed not only for the encrypted data, but also to store information containing attributes of the encryption algorithm and encrypted data (for example, CCSID). The data stored in these extra bytes allows DB2 for i5/OS to share and exchange encrypted data with other DB2 server products. A minimum of 8 extra bytes for RC2 encryption or 16 bytes for TDES encryption are needed for these attributes. If the data string is defined as large object (LOB), BINARY, or VARBINARY, or if any of the data strings, hints, or passwords have different CCSID values, then 8 additional bytes are required, making 16 extra bytes for RC2 encryption or 24 extra bytes for TDES encryption.

If a hint is specified along with the password used for encrypting the data, 32 additional bytes are added to the column length. If no hint is specified, no additional bytes are added to the encrypted column length for the hint. Finally, the column length is padded to make the entire column length an even multiple of 8 bytes. So the column length of a column containing encrypted data is:

$\text{data-length} + \text{extra-bytes} + \text{hint-length} + \text{pad}$

In our example, `ssn` and `salary` require that we increase the size and type of these columns in order to store the encrypted data. Because we are encrypting non-binary data using the TDES algorithm, 16 extra bytes are required. Assuming our hint size to be 32 (maximum), then the sizes of our encrypted columns become:

```
ssn length is
9 + 16 + 32 + pad
57 + 7
64
```

```
salary length is
13 + 16 + 32 + pad
61 + 3
64
```

Therefore, our new table structure is now:

```
CREATE TABLE empTDES (
  employeeId CHAR(7),
  ssn CHAR(64) FOR BIT DATA
  name VARCHAR(40),
  salary CHAR(64) FOR BIT DATA
)
```

### 10.1.3 Analyzing impact to performance

Encryption of data that is frequently referenced or used in a lookup operation may result in slower performance. Encrypting columns (or fields) may also make sorting or searching on them difficult.

Encryption should be reserved only for data that is personal or sensitive. Minimizing the columns (or fields) that require encryption reduces the impact to performance greatly. If a column used as a search key must be encrypted, then encrypting the query's search value allows the search to be conducted without first decrypting all the rows of the table. Searching on an encrypted value is also better for security than having to decrypt all rows of the table to accomplish the search. However, sorting an encrypted column is not meaningful. The column must be decrypted to be sorted properly, or an index maintained to preserve the plaintext sort order.

In our example, `ssn` and `salary` are not used primarily for table lookup or for sorting, nor are they used as an index to another table, so the impact of encryption of these fields to our application performance should be manageable without significant changes to the structure of our application.

## 10.2 Encrypting data using an encryption password

Use the SET ENCRYPTION PASSWORD statement to set the default password for the encryption and decryption functions. The key used by SQL for encrypting and decrypting is not the password value, but a 128-bit binary value automatically derived from the password using an MD5 message digest.

The password is type CHAR, VARCHAR, Unicode GRAPHIC, or Unicode VARGRAPHIC and must be between 6 and 127 characters in length, or must be an empty string. If an empty string is specified, the default encryption password is set to no value. For example:

```
SET ENCRYPTION PASSWORD='enig1942ma'
```

We insert the numeric digits in the middle of the string 'enigma' in order to make the password harder to guess. A hacker could use a program to attempt decryption using words from the dictionary, and could guess the string 'enigma' easier than the string 'enig1942ma'. Also, a longer password is more secure than a short password. Refer to 3.3.1, "Generating a key value" on page 28, for more information about forming secure keys.

It is preferable to specify the password using a host variable rather than a string constant so that access to the source code does not compromise security. For example:

```
SET ENCRYPTION PASSWORD=:passwordvar
```

When using a remote relational database, the password specified is sent as plaintext. In order to protect the supplied password, the communication channel must be secured by encryption such as SSL to maintain security.

### 10.2.1 Associating a hint with a password

Use the WITH HINT clause to specify a value that will help to recall the password value when the password is lost or forgotten.

The hint is type CHAR, VARCHAR, Unicode GRAPHIC, or Unicode VARGRAPHIC, and must not be greater than 32 characters in length. If an empty string is specified, the default encryption password hint is set to no value.

If a WITH HINT value is specified, the hint is used as the default during encryption functions. The hint value is stored together with encrypted data as a reminder of the associated password. The hint can be retrieved for an encrypted value using the GETHINT function. To demonstrate, we create an encrypted employee record after assigning a hint value:

```
CREATE TABLE empTDES (  
  employeeId CHAR(7),  
  ssn CHAR(64) FOR BIT DATA  
  name VARCHAR(40),  
  salary CHAR(64) FOR BIT DATA
```

```
SET ENCRYPTION PASSWORD='enig1942ma' WITH HINT='submarine cipher'
```

```
INSERT INTO empTDES VALUES(  
  '1000001',  
  ENCRYPT_TDES('111223333'),  
  'ROBERT',  
  ENCRYPT_TDES(4000000))
```

Since the WITH HINT value was the same for encrypting both the ssn and the salary of the emp record, the GETHINT function returns the same hint value for either of the two encrypted fields:

```
SELECT GETHINT(ssn) FROM empTDES
submarine cipher
```

```
SELECT GETHINT(salary) FROM empTDES
submarine cipher
```

It is preferable to assign the WITH HINT value using a host variable rather than a string constant so that access to the source code does not compromise security. For example:

```
SET ENCRYPTION PASSWORD=:passwordvar WITH HINT=:hintvar
```

When using a remote relational database, the hint specified is sent as plaintext. In order to protect the supplied hint, the communication channel must be secured by encryption, such as SSL, to maintain security.

## 10.2.2 Using a password in a view

A simple method to access the decrypted data using a password is with a VIEW:

```
SET ENCRYPTION PASSWORD=:passwordvar
CREATE VIEW empview(employeeId, ssn, name, salary) AS
  SELECT
    employeeId,
    char(DECRYPT_CHAR(ssn), 9 ),
    name,
    char(DECRYPT_CHAR(salary), 13)
  FROM empTDES
```

This convenient approach allows decrypted data to be retrieved when the password is provided.

Allowing access to the entire table with a single password may be convenient, but may not be desirable from a security standpoint. Should the single password become compromised, all records in the table are accessible. Data security must be reviewed carefully when using this approach to decryption.

## 10.2.3 Using password and hint as encryption parameters

We have already seen how to use the SET ENCRYPTION PASSWORD to assign the password (and hint) for the entire table. We can alternatively supply the password (and hint) using the SQL built-in encryption function parameters. This method allows us to specify different passwords (and hints) for each table record.

```
INSERT INTO empTDES VALUES(
  '1000001',
  ENCRYPT_TDES('111223333',:passwordvar,:hintvar),
  'ROBERT',
  ENCRYPT_TDES(4000000,:passwordvar,:hintvar))
```

Supplying different passwords for each record in the table could be used to allow a customer to supply a password and hint for her own account information, for example. Then if the password is discovered, only a single record is compromised rather than the entire table.

## 10.3 Encrypting data with triggers

We can use triggers to encrypt and decrypt data. A trigger allows writes and updates to be intercepted for encryption without impact to the applications accessing the database. It does not matter whether the access to the data is performed by SQL or a native language application. The trigger performs the same, regardless of the programming language of the application.

### 10.3.1 Using classical triggers

Use a BEFORE INSERT trigger to encrypt data from a native WRITE or SQL Insert. First, the user-defined function GET\_PASSWD() is called to get the encryption password for the employee record. The password is then used to encrypt the sensitive data before it is stored.

```
CREATE TRIGGER insert_empTDES
  BEFORE INSERT ON empTDES
  REFERENCING NEW ROW AS n
  FOR EACH ROW
  BEGIN
    DECLARE encrypt_passwd VARCHAR(127);
    SET encrypt_passwd = GET_PASSWD(:hintvar);
    SET n.ssn = ENCRYPT_TDES(n.ssn, encrypt_passwd);
    SET n.salary = ENCRYPT_TDES(n.salary, encrypt_passwd);
  END
```

Use an UPDATE trigger in a similar fashion to encrypt data from a native or SQL UPDATE.

### 10.3.2 Using Instead Of Triggers

Another type of trigger that may be used is known as an Instead Of Trigger. This type of trigger may be used over an SQL view to encrypt data for insert, update, and delete operations.

Instead Of Triggers are useful in reducing the impact of encryption to your applications. The view takes the place of the unencrypted table, and the Instead of Triggers do the work of mapping the unencrypted columns to the encrypted ones.

In our example, triggers are created over the view empView in order to insert and update to table empTDES.

```
CREATE TRIGGER insert_empView
  INSTEAD OF INSERT ON empView
  REFERENCING NEW AS n
  FOR EACH ROW MODE DB2SQL
  INSERT INTO empTDES VALUES(
    n.employeeId,
    ENCRYPT_TDES(n.ssn, :passwordvar, :hintvar),
    n.name,
    ENCRYPT_TDES(n.salary, :passwordvar, :hintvar)
  )

CREATE TRIGGER update_empView
  INSTEAD OF UPDATE ON empView
  REFERENCING OLD AS o NEW AS n
  FOR EACH ROW MODE DB2SQL
```

```

UPDATE empTDES SET
    employeeId=n.employeeId,
    ssn=ENCRYPT_TDES(n.ssn, :passwordvar, :hintvar),
    name=n.name,
    salary=ENCRYPT_TDES(n.salary, :passwordvar, :hintvar)
WHERE employeeId=o.employeeId

```

## 10.4 Using user-defined functions (UDFs) with encrypted data

Additional capabilities not available in SQL statements can be provided by using user-defined functions. We can use the CREATE FUNCTION statement to provide functions that may be referenced in SQL statements.

In this example we show how to create a user-defined function in the SQL language in order to search for the employee record matching the value of ssnIndex. The function encrypts the ssnIndex value using the currently set encryption password and searches on the encrypted ssn field of the table to return the employeeId of the matching record. This method is more efficient and also more secure than decrypting the ssn field of each record in the table and comparing the plaintext values because in this method the encryption function only executes once on the ssnIndex. The encryption password value must be set before the function is called because the SET ENCRYPTION PASSWORD statement cannot be performed inside a user-defined function. If we wished to set the password as a statement within the routine, we could create the routine as a stored procedure rather than a user-defined function.

```

CREATE FUNCTION LIB01+/FIND_SSN(
    ssnIndex CHAR(9))
    RETURNS CHAR(7)
    LANGUAGE SQL
    SPECIFIC FIND_SSN NOT DETERMINISTIC
    NO EXTERNAL ACTION READS SQL DATA
    RETURNS NULL ON NULL INPUT
    ALLOW PARALLEL
    RETURN(
    SELECT employeeId FROM empTDES
    WHERE ssn=ENCRYPT_TDES(ssnIndex)
    )

```

**Note:** User-defined functions that reference SQL built-in functions for encryption and decryption must be defined as NOT DETERMINISTIC. This ensures that if a SET ENCRYPTION PASSWORD statement changes the password, the new password value will be used by the UDF.

In 10.3.1, “Using classical triggers” on page 120, we indicated that a user-defined function could be written to return a password value, which is then used to encrypt or decrypt record data. In this way the details of managing the password can be separated from the application code working with the encrypted data.

In this example we provide two user-defined functions written in the C programming language. The GET\_PASSWD function accepts the hint value as a parameter and uses it as a label to retrieve the associated password from the HINTLKUP table. The PUT\_PASSWD function is used to save a new password along with the associated hint in the HINTLKUP table. These user-defined functions use the C language encryption and decryption functions so that the password is stored in the HINTLKUP table as ciphertext and not as plaintext data.

Note that since we are using the native encryption and decryption functions in C instead of the SQL built-in functions to do our enciphering, we need not add the extra bytes for attributes (for example, CCSID), but only pad to the next 8-byte boundary when sizing the ciphertext buffer. The GET\_PASSWD and PUT\_PASSWD user-defined functions reference a key from the keystore file TDESKEYS when encrypting and decrypting the password. We show how the key and the keystore file were generated later in 10.5, “Encrypting with stored procedures” on page 127

The HINTLKUP table is described below in DDS. This data is accessed in our C language UDF as a physical file, but can also be accessed from SQL statements as a table. The HINT field contains the plaintext hint and serves as the file index key. The PASSWD field contains ciphertext encrypted with the TDES algorithm. The PASSWDIV field is used to store the Initialization Vector, and is needed to decrypt the PASSWD ciphertext. The KEYLABEL field identifies the encryption key in the TDESKEYS keystore used to encipher the PASSWD field.

```

A          R RFHINTLKUP          TEXT('Hint Lookup Record')
A*
A          HINT          32A      TEXT(' Hint for password')
A*
A          PASSWD        128A     TEXT('Password')
A*
A          PASSWDIV      8A       TEXT('Password IVector')
A*
A          KEYLABEL      32A      TEXT('Key Label')
A*
A          K HINT

```

Do not forget to use object authorities to restrict access to this table in order to ensure that the password information it contains is secure. Refer to 5.4, “Establishing a secure keystore environment” on page 65, for details on securing objects containing key information.

The C code that implements the user-defined functions is shown below. Since the GET\_PASSWD function decrypts the password using whatever key is identified by the label value KEYLABEL from the HINTLKUP table, there is no need to re-encrypt the password entries when the key used in PUT\_PASSWD changes, so long as the former key remains in the TDESKEYS keystore identified with its label. Before a key from the keystore is deleted or destroyed, all the records in HINTLKUP (and any other tables referencing the key label) for that key should be decrypted and encrypted using the newest key.

```

/*****/
/*
/* CRTCMOD MODULE(LIB01/UDF_PASSWD)
/* SRCFILE(LIB01/QCSRC)
/* TEXT('Password UDF functions')
/* OUTPUT(*PRINT) OPTION(*SHOWINC) DBGVIEW(*LIST)
/*
/*
/* CRTSRVPGM SRVPGM(LIB01/UDF_PASSWD)
/* MODULE(LIB01/UDF_PASSWD)
/* EXPORT(*ALL) ACTGRP(*CALLER)
/*
/*
/*****/

#include <stdio.h>          /* Standard I/O library    */
#include <stdlib.h>        /* Standard C library      */
#include <recio.h>         /* Record I/O library      */
#include <string.h>        /* String library          */

```



```

#include <qusec.h> /* Error code structure */
#include <stddef.h> /* Standard C library */
#include <sqludf.h> /* SQL User-Defined Functions */
#include <qc3cci.h> /* Crypto Common library */
#include <qc3prng.h> /* Crypto Pseudo-Random library */
#include <qc3dtaen.h> /* Crypto Encrypt Data library */
#include <qc3dtade.h> /* Crypto Decrypt Data library */

#pragma mapinc("HINTLKUP","HINTLKUP(*ALL)","both key","d z _P",,"DDS")
#include "HINTLKUP"

#define MAX_HINT 32
#define MAX_PASSWD 127
#define MAX_CIPHER 128

_RFILE *fp_Hintlkup = NULL;

/*****
/*
/* Function:
/* GET_PASSWD
/*
/* Parameters:
/* hint VARCHAR(32) User supplied hint
/*
/* Returns:
/* passwd VARCHAR(127) Password value associated with hint
/*
/* Description:
/* This function receives the hint parameter and returns an
/* associated password from file HINTLKUP.
/*
*****/
void SQL_API_FN GET_PASSWD (
    SQLUDF_VARCHAR hint[MAX_HINT+1],
    SQLUDF_VARCHAR passwd[MAX_PASSWD+1],
    SQLUDF_NULLIND *hintInd,
    SQLUDF_NULLIND *passwdInd,
    SQLUDF_TRAIL_ARGS)
{
    Qus_EC_t errCode; /* Error code structure */
    _RIOFB_T *iofb_Hintlkup;
    DDS_RFHINTLKUP_both_t rfHintlkup;
    DDS_RFHINTLKUP_key_t rfHintlkup_key;
    Qc3_Format_ALGD0200_T algDesc;
    Qc3_Format_KEYD0400_T keyDesc;
    char csp = Qc3_Any_CSP;
    int maxPasswdLen;
    int cipherLen;
    int passwdLen;

    memset(&errCode, 0, sizeof(errCode));
    memset(passwd, '\0', MAX_PASSWD + 1);
    passwdLen = 0;
    maxPasswdLen = MAX_PASSWD + 1;

```

```

cipherLen = MAX_CIPHER;

if (fp_Hintlkup == NULL)
{
    fp_Hintlkup = _Ropen("HINTLKUP", "rr+");
    if (fp_Hintlkup == NULL)
    {
        printf("Open HINTLKUP failed\n");
        return;
    }
}

memset(rfHintlkup_key.HINT, '\0', sizeof(rfHintlkup_key.HINT));
memcpy(rfHintlkup_key.HINT, hint, strlen(hint));
iofb_Hintlkup = _Rreadk(fp_Hintlkup, &rfHintlkup, sizeof(rfHintlkup),
    __DFT | __NO_LOCK, &rfHintlkup_key, sizeof(rfHintlkup_key));
if (iofb_Hintlkup->num_bytes != sizeof(rfHintlkup))
{ /* Read Failed */
    printf("Read HINTLKUP failed, hint value %s not found\n", hint);
}
else
{ /* Read Successful */

    /* Prepare for passwd Decryption using TDES Algorithm */

    /* Algorithm Description */
    memset(&algDesc, '\0', sizeof(algDesc));
    algDesc.Block_Cipher_Alg = Qc3_TDES;
    algDesc.Block_Length = 8;
    algDesc.Mode = Qc3_CBC;
    algDesc.Pad_Option = Qc3_Pad_Char;
    algDesc.Pad_Character = '\0';
    algDesc.MAC_Length = 0;
    algDesc.Effective_Key_Size = 0;
    memcpy(&algDesc.Init_Vector[0], &rfHintlkup.PASSWDIV[0], 8);

    /* Key Description */
    memset(&keyDesc, '\0', sizeof(keyDesc));
    memcpy(&keyDesc.Key_Store[10], "JOHNC      ", 10);
    memcpy(&keyDesc.Key_Store[0], "TDESKEYS  ", 10);
    memcpy(&keyDesc.Record_Label[0], &rfHintlkup.KEYLABEL[0], 32);

    /* Decrypt PASSWD ciphertext into passwd plaintext */
    Qc3DecryptData(&rfHintlkup.PASSWD[0], &cipherLen,
        (char *) &algDesc, Qc3_Alg_Block_Cipher,
        (char *) &keyDesc, Qc3_Key_KSLabel,
        &csp, NULL,
        &passwd[0], &maxPasswdLen,
        &passwdLen, &errCode);
}
if ( errCode.Bytes_Available != 0 )
{
    printf("Data Decryption of password failed, Exception_Id = %.7s\n",
        errCode.Exception_Id);
    return;
}

```

```

    }

    return;

}

/*****
/*
/* Function:
/* PUT_PASSWD
/*
/* Parameters:
/* hint VARCHAR(32) User supplied hint
/* passwd VARCHAR(127) Password value associated with hint
/*
/* Returns:
/* rsltlen DOUBLE CAST TO INT Encrypted password length
/*
/* Description:
/* This function stores the hint parameter and associated
/* password in file HINTLKUP and returns the length of the
/* encrypted password.
/*
*****/
void SQL_API_FN PUT_PASSWD (
    SQLUDF_VARCHAR hint[MAX_HINT+1],
    SQLUDF_VARCHAR passwd[MAX_PASSWD+1],
    SQLUDF_DOUBLE *rsltlen,
    SQLUDF_NULLIND *hintInd,
    SQLUDF_NULLIND *passwdInd,
    SQLUDF_NULLIND *rsltlenInd,
    SQLUDF_TRAIL_ARGS)
{
    Qus_EC_t errCode; /* Error code structure */
    _RIOFB_T *iofb_Hintlkup;
    DDS_RFHINTLKUP_both_t rfHintlkup;
    DDS_RFHINTLKUP_key_t rfHintlkup_key;
    char PRNType = Qc3PRN_TYPE_NORMAL;
    char PRNParity = Qc3PRN_NO_PARITY;
    Qc3_Format_ALGD0200_T algDesc;
    Qc3_Format_KEYD0400_T keyDesc;
    char csp = Qc3_Any_CSP;
    int passwdLen;
    int maxCipherLen = sizeof(rfHintlkup.PASSWD);
    int cipherLen;

    memset(&errCode, 0, sizeof(errCode));
    errCode.Bytes_Provided = sizeof(errCode);
    *rsltlen = 0;
    cipherLen = 0;

    if (fp_Hintlkup == NULL)
    {
        fp_Hintlkup = _Ropen("HINTLKUP", "rr+");
        if (fp_Hintlkup == NULL)

```

```

    {
        printf("Open HINTLKUP failed\n");
        return;
    }
}

memset(rfHintlkup.HINT, '\0', MAX_HINT);
memcpy(rfHintlkup.HINT, hint, strlen(hint));

/* Prepare to Encrypt passwd with TDES */
/* Pad password with NULLS and change length to encrypt maximum password */
passwdLen = strlen(passwd);
memset(&passwd[passwdLen], '\0', MAX_PASSWD + 1 - passwdLen);
passwdLen = MAX_PASSWD;
memset(rfHintlkup.PASSWD, '\0', MAX_CIPHER);

/* Algorithm Description */
memset(&algDesc, '\0', sizeof(algDesc));
algDesc.Block_Cipher_Alg = Qc3_TDES;
algDesc.Block_Length = 8;
algDesc.Mode = Qc3_CBC;
algDesc.Pad_Option = Qc3_Pad_Char;
algDesc.Pad_Character = '\0';
algDesc.MAC_Length = 0;
algDesc.Effective_Key_Size = 0;
/* Generate Random Initialization Vector for PASSWD field */
Qc3GenPRNs(&rfHintlkup.PASSWDIV[0], 8, PRNType, PRNParity, &errCode);
if (errCode.Bytes_Available != 0)
{ /* Pseudo-Random Number Generation Failed */
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}
memcpy(&algDesc.Init_Vector[0], &rfHintlkup.PASSWDIV[0], 8);

/* Key Description */
memset(&keyDesc, '\0', sizeof(keyDesc));
memcpy(&keyDesc.Key_Store[10], "KYLIB      ", 10);
memcpy(&keyDesc.Key_Store[0], "TDESKEYS  ", 10);
memset(&keyDesc.Record_Label[0], '\0', 32);
memcpy(&keyDesc.Record_Label[0], "PASSWDKEY.1", 11);

memcpy(rfHintlkup.KEYLABEL, &keyDesc.Record_Label[0],
    sizeof(keyDesc.Record_Label));
Qc3EncryptData(&passwd[0], &passwdLen, Qc3_Data,
    (char *)&algDesc, Qc3_Alg_Block_Cipher,
    (char *)&keyDesc, Qc3_Key_KSLabel,
    &csp, NULL,
    &rfHintlkup.PASSWD[0], &maxCipherLen,
    &cipherLen, &errCode);
if ( errCode.Bytes_Available != 0 )
{
    printf("Data Encryption of password failed, Exception_Id = %.7s\n",
        errCode.Exception_Id);
    return;
}
}

```

```

iofb_Hintlkup = _Rwrite(fp_Hintlkup, &rfHintlkup, sizeof(rfHintlkup));
if (iofb_Hintlkup->num_bytes != sizeof(rfHintlkup))
{
    printf("Write HINTLKUP failed\n");
}
else
{
    *rsltlen = cipherLen;
}
return;
}

```

Once the service programs are created, the UDFs are then registered with SQL. The functions can then be invoked within SQL statements.

The user-defined functions are registered in SQL using the following statements:

```

CREATE FUNCTION LIB01/GET_PASSWD(
hint VARCHAR(32))
RETURNS VARCHAR(127)
LANGUAGE C
PARAMETER STYLE SQL
SPECIFIC GET_PASSWD
NOT DETERMINISTIC
NO SQL
NO EXTERNAL ACTION
EXTERNAL NAME 'LIB01/UDF_PASSWD(GET_PASSWD) '

CREATE FUNCTION LIB01/PUT_PASSWD(
hint VARCHAR(32),
passwd VARCHAR(127))
RETURNS INT CAST FROM FLOAT
LANGUAGE C
PARAMETER STYLE SQL
SPECIFIC PUT_PASSWD
NOT DETERMINISTIC
NO SQL
NO EXTERNAL ACTION
EXTERNAL NAME 'LIB01/UDF_PASSWD(PUT_PASSWD) '

```

## 10.5 Encrypting with stored procedures

Stored procedures can be used to enhance and simplify application operations.

In this example, we use a stored procedure in the SQL language to show how to retrieve and set the encryption password, then encrypt the data for record update in one application operation. We use the GETHINT built-in function to retrieve the hint associated with the password that was used to encrypt the existing record. The hint is passed to the GET\_PASSWD user-defined function, which uses the hint as a label to return the corresponding encryption password. The procedure then sets the encryption password with the value returned from the UDF function, and updates the record with the newSalary data encrypted using the password.

```
CREATE PROCEDURE LIB01/UPDATE_SALARY(
employeeIdKey CHAR(7),
newSalary NUMERIC(13))
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
DECLARE hint VARCHAR(32);
DECLARE passwd VARCHAR(127);
SELECT GETHINT(salary) INTO hint
  FROM empTDES WHERE employeeId = employeeIdKey;
SET passwd = GET_PASSWD(hint);
SET ENCRYPTION PASSWORD=passwd WITH HINT hint;
UPDATE empTDES
  SET salary = ENCRYPT_TDES(CHAR(newSalary))
  WHERE employeeId = employeeIdKey;
END
```

Like external UDFs, stored procedures can be created using native languages and called from SQL once they are registered.

We show with this example a stored procedure that is used to generate an encryption key and save the key in the TDESKEYS keystore file. The UDFs GET\_PASSWD and PUT\_PASSWD use this key when encrypting and decrypting data in the HINTLKUP table. This key could be shared by other programs or applications as well.

Notice that the key value itself is not observable either by the user or the programmer in this instance. Rather, the key is identified and referenced by a label maintained in the keystore and the HINTLKUP table data.

We assume that the system administrators have already generated a master key 1, which is referenced when creating the TDESKEYS keystore. For information about creating master keys, refer to 5.1.1, “Master keys” on page 48.

```
/******  
/*  
/* CRTCMOD MODULE(LIB01/CRT_TDES)  
/* SRCFILE(LIB01/QCSRC)  
/* TEXT('Create TDES Encryption Key')  
/* OUTPUT(*PRINT) OPTION(*SHOWINC) DBGVIEW(*LIST)  
/*  
/* CRTBND CPGM(LIB01/CRT_TDES)  
/* SRCFILE(LIB01/QCSRC) SRCMBR(CRT_TDES)  
/* OUTPUT(*PRINT) OPTION(*SHOWINC) DBGVIEW(*LIST)  
/* REPLACE(*YES)  
/*  
/******  
  
#include <stdio.h> /* Standard I/O library */
```

```

#include <stdlib.h>          /* Standard C library      */
#include <recio.h>          /* Record I/O library      */
#include <string.h>         /* String library          */
#include <qusec.h>          /* Error code structure    */
#include <stddef.h>        /* Standard C library      */
#include <qc3cci.h>        /* Crypto Common library   */
#include <qc3crtks.h>     /* Crypto Create Key Store */
#include <qc3krgen.h>     /* Crypto Generate Key Record */
#include <sys/stat>        /* Get File Information    */

#define RCOK                0
#define RCERROR            -1
#define KEYSTORELIB        "KYLIB  "
#define KEYSTORENAME      "TDESKEYS "
/*****
/*
/* Procedure:
/* CRT_TDES
/*
/* Parameters:
/* In label CHAR(32) Key identification label
/*
/* Description:
/* This program generates a TDES encryption key and stores
/* it with the associated label in the keystore file TDESKEYS.
/* If the keystore file does not exist, it is created under
/* Master Encryption Key 1.
/*
*****/
int main (int argc, char *argv[])
{
    /* Error codes */
    int rc;          /* Return code, 0 = successful */
                    /* -1 = error */

    Qus_EC_t        errCode; /* Error code structure */

    /* Parameters */
    char            keyLabel[32]; /* Input, label id of key */

    /* Keystore Information */
    int             nameIndex;
    char            keystoreLib[10];
    char            keystoreName[10];
    char            keystorePath[41];
    char            keystoreQualName[20];
    struct stat     fileInfo;

    /* Create Keystore Parameters */
    int             masterKeyId;
    char            keystoreAuth[10];
    char            keystoreDesc[50];

    /* Generate Key Record Parameters */
    int             krecType;

```

```

int          krecSize;
int          krecExponent;
int          krecDisallow;
char         krecServProv;
char         krecDevName[10];

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
/* Assign parameter to local variable */
strncpy(keyLabel, argv[1], sizeof(keyLabel));

/* Initialize keystore Qualified File name */
memcpy(&keystoreQualName[0], KEYSTORENAME, sizeof(keystoreName));
memcpy(&keystoreQualName[10], KEYSTORELIB, sizeof(keystoreLib));

/* Initialize keystore Path name */
memcpy(keystoreLib, KEYSTORELIB, sizeof(keystoreLib));
nameIndex = sizeof(keystoreLib) - 1;
while ((nameIndex >= 0) && (keystoreLib[nameIndex] == ' '))
    keystoreLib[nameIndex--] = '\\0';
memcpy(keystoreName, KEYSTORENAME, sizeof(keystoreName));
nameIndex = sizeof(keystoreName) - 1;
while ((nameIndex >= 0) && (keystoreName[nameIndex] == ' '))
    keystoreName[nameIndex--] = '\\0';
memset(keystorePath, '\\0', sizeof(keystorePath));
sprintf(keystorePath, "/QSYS.LIB/%.10s.LIB/%.10s.FILE",
        keystoreLib, keystoreName);

/* Check if keystore exists */
if (0 != stat(&keystorePath[0], &fileInfo))
{ /* Keystore not found, create it */
    masterKeyId = 1;
    memcpy(keystoreAuth, "*EXCLUDE ", sizeof(keystoreAuth));
    memset(keystoreDesc, ' ', sizeof(keystoreDesc));
    memcpy(keystoreDesc, "Key store for TDES keys", 23);

    Qc3CreateKeyStore(keystoreQualName, &masterKeyId,
        keystoreAuth, keystoreDesc, &errCode);
    if (errCode.Bytes_Available != 0 )
    {
        printf("CRT_TDES: Create keystore failed, Exception_Id = %.7s\\n",
            errCode.Exception_Id);
        rc = RCERROR;
        return rc;
    }
}

/* Create TDES key */
krecType = Qc3_TDES;
krecSize = 24;
krecExponent = 0; /* Not used for TDES */
krecDisallow = 0; /* No functions disallowed */
krecServProv = Qc3_Any_CSP; /* Software or Hardware */
memset(krecDevName, ' ', sizeof(krecDevName)); /* Not used for TDES */
Qc3GenKeyRecord(keystoreQualName, keyLabel, &krecType, &krecSize,

```



```
    &krecExponent, &krecDisallow, &krecServProv, krecDevName, &errCode);
if (errCode.Bytes_Available != 0 )
{
    printf("CRT_TDES: Create key record failed, Exception_Id = %.7s\n",
        errCode.Exception_Id);
    rc = RCERROR;
    return rc;
}

rc = RCOK;
return rc;
}
```





## Cryptographic Services APIs method

The Cryptographic Services APIs perform cryptographic function within i5/OS or on the 2058 Cryptographic Accelerator.

The Cryptographic Services APIs include:

- ▶ Encryption and Decryption APIs
- ▶ Authentication APIs
- ▶ Key Generation APIs
- ▶ Key Management APIs
- ▶ Pseudorandom Number Generator APIs
- ▶ Cryptographic Context APIs

A sample application utilizing these APIs is provided in this Redbooks publication. This chapter provides detailed information about that sample application and its scenario.

## 11.1 Scenario description

Using a practical scenario, we walk through the process of using the Cryptographic Services APIs to protect private data.

We have an application that processes customer data and, due to the sensitive nature of some of the data elements, we have decided to use encryption. The basic structure of the scenario is outlined in Figure 11-1.

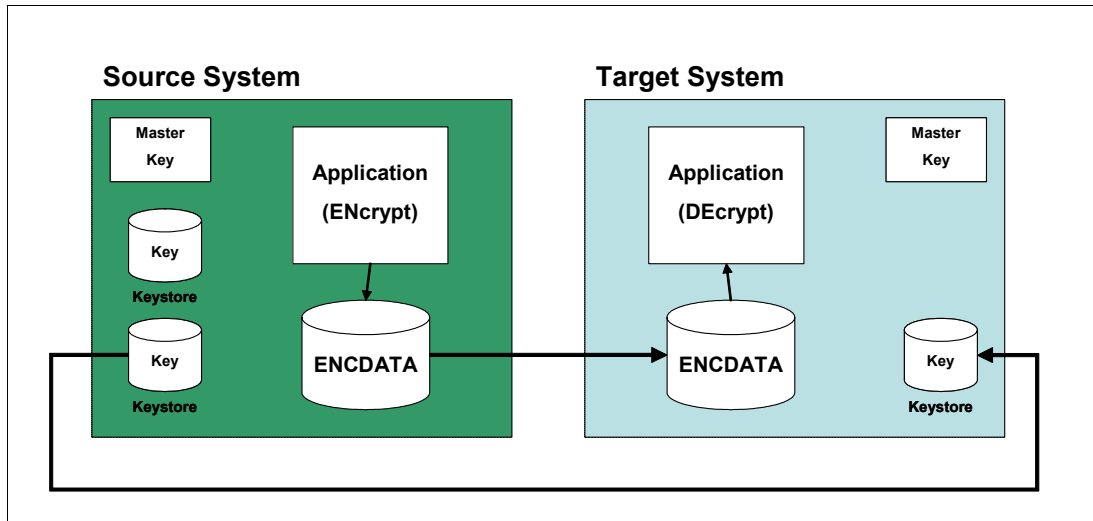


Figure 11-1 Cryptographic Services APIs scenario description

### 11.1.1 Setting up a master key

The first step in the application process is to create a master key. The master key is used to encrypt a data encryption key in a keystore file.

The sample application provides a CL command named Set Master Key (SET\_MSTR\_K). This command performs the creation of the master key. The process of creating the master key using this command is illustrated in Figure 11-2.

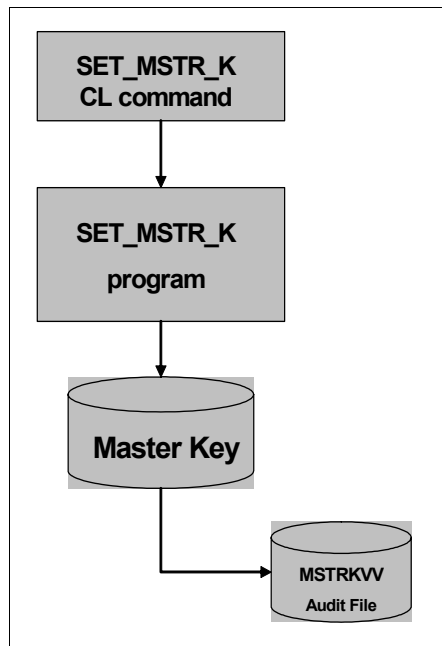


Figure 11-2 Creating a master key using SET\_MSTR\_K

## 11.1.2 Setting up a symmetric data encryption key

After the master key is created, we create a symmetric data encryption key. A symmetric key is the type of key where the same key is used to perform both the encryption and decryption of data. Although it may seem slightly confusing, the term *data encryption key* refers to a key that has the potential of performing encryption or decryption functions on data.

The data encryption key, often referred to simply as the *key*, can be stored in a special type of i5/OS object called a keystore file. All key values in a keystore file are encrypted under a master key. The data encryption key record in the keystore file is referenced by a unique name called a *label*. The key record also includes information about the functional capabilities of the key. The key that we create will be configured to allow it perform both encryption and decryption functions.

Our scenario entails an application that uses the data encryption key to encrypt user-supplied information and store the result in a file. There is also a second application on a separate server that is permitted to only decrypt data. Because of this, we also create a second keystore file. In the second keystore file, we store a duplicate of the data encryption key.

The reason for the requirement of a second keystore file is that we need a key with the same key label but different functional capabilities (we are going to permit this key to only perform decryption functions). The second keystore is copied over to the target system for use by the decrypting application.

**Note:** If the second application was permitted to perform both encryption and decryption, then we would only require one keystore file with a single key record. Alternatively, if the second application referenced the decrypting-only key by a different label, then we could store it in the same keystore file as the full-function key.

The sample application also provides a CL command named Generate Symmetric Key (GEN\_SYMKEY). This command performs the creation of the two keystore files and the two copies of the symmetric key. The process of creating the symmetric key using the command is depicted in Figure 11-3.

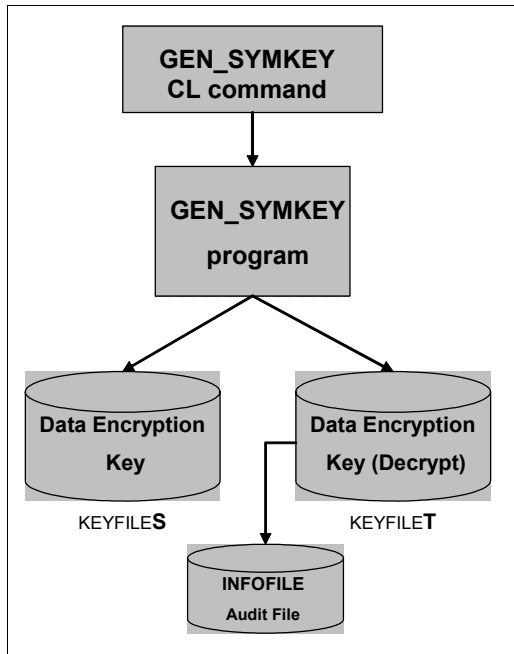


Figure 11-3 Creating a symmetric key using GEN\_SYMKEY

### 11.1.3 Encrypting data

Using the encryption-capable data encryption key, the application will take user-supplied data and encrypt it before storing it in a database file (ENCDATA).

The sample application also provides a CL command named Set Customer Data (SET\_DATA). This command performs the encryption of data. The process of encrypting the customer data using the command is depicted in Figure 11-4.

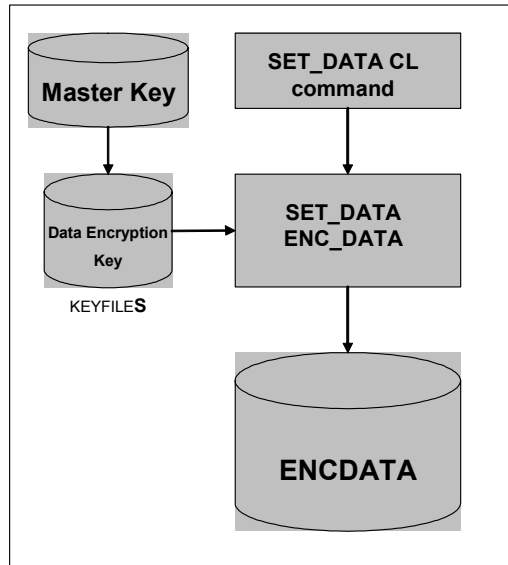


Figure 11-4 Encrypting customer data using SET\_DATA

### 11.1.4 Decrypting data

Now that the data includes some ciphertext, we need a mechanism to retrieve the plaintext again.

Depending on whether the decryption is going to take place on the source or on the target system, we have different procedures.

#### Decryption of data on source system

We use this scenario if we want to encrypt/decrypt the secret data in the database file on the same system.

#### Decryption of data on target system

The difference from decrypting on the source system comes down to the key maintenance. The decrypting application must have access to the keystore file that contains the keys used to encrypt the secret information in the customer's data.

After the database file and the keystore file containing the decrypting-only key are transferred across a secure channel (for example, SSL or Secure FTP) to the target system, we can take the ciphertext and return it to its plaintext state.

## 11.1.5 Scenario analysis and summary of APIs used

Figure 11-5 illustrates the sample application provided for this chapter. A detailed description of this diagram is provided in following sections.

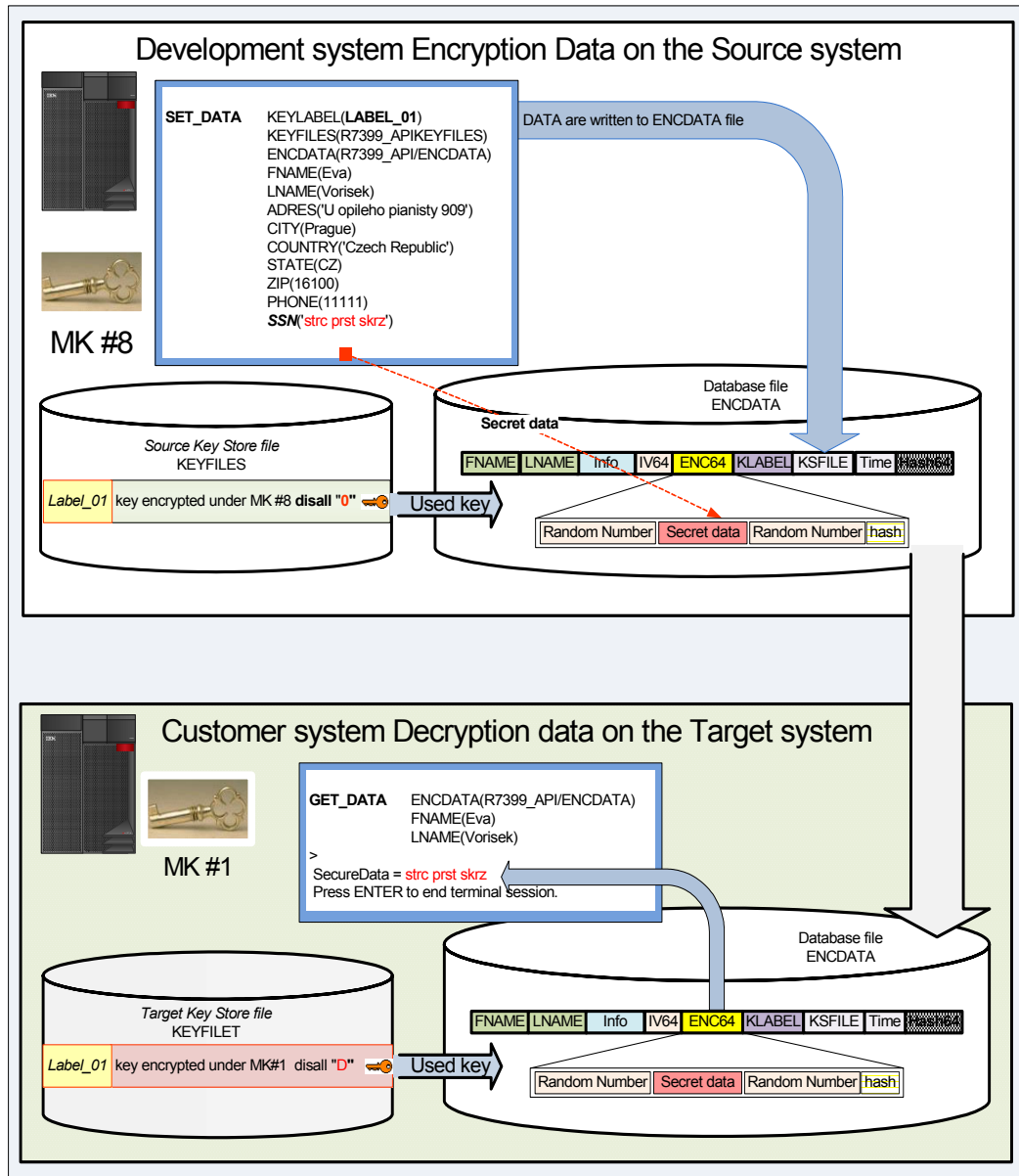


Figure 11-5 Scenario diagram

### The APIs used by the scenario application

Our scenario shows you how to use several APIs, including:

- ▶ Encryption/Decryption APIs

These APIs allow you to transform information into unintelligible data, thus allowing you to store sensitive data more securely. They also allow you to transform the unintelligible data back into human readable form.

- ▶ Authentication APIs

The authentication APIs help you ensure that the data has not been altered.



- ▶ Key Management APIs

These APIs allow you to clear, load, and set master keys, as well as generate keys and create special database files called keystores, used for storing cryptographic keys.

- ▶ Pseudorandom Number Generator API

This API is used to generate a random binary string that can be used for many purposes, such as an Initialization Vector (IV) for the Encryption/Decryption APIs.

- ▶ Cryptographic context APIs

These APIs allow you to temporarily store the key and algorithm parameters to avoid repeated retrieval during subsequent cryptographic operations, and to maintain the state of the operation when performed over multiple calls.

We also show how to perform the translation process for keys encrypted under the master key, when the master key is changed.

## 11.2 Scenario application setup

Before we can use the application, we need to create the application objects for the various functions performed by the application. All programs and their source code referenced in this chapter are created and available from the IBM Redbooks Web site described in Appendix A, “Additional material” on page 281.

### 11.2.1 Sample application download and initial setup

All required objects, such as programs, commands, and their source code, are stored in the library R7399\_API.

1. Download the save file R7399\_API.savf of the library R7399\_API from the Web site as described in Appendix A, “Additional material” on page 281.
2. The initial setup for our scenario is:
  - a. Restore the library R7399\_API with the command:

```
RSTLIB SAVLIB(R7399_API) DEV(*SAVF) SAVF(QGPL/R7399_API)
```
  - b. Add this library to the library list with the command:

```
ADDLIBLE R7399_API
```

### 11.2.2 Creating commands for sample application scenario

As part of the sample application, several commands are provided to simplify the invocation of the various cryptographic APIs. The following sections provide set-by-step guides for creating these commands.

#### Master key creation (SET\_MSTR\_K)

To build this command, create the following objects:

1. Physical file MSTRKVV to hold the key validation values:

```
CRTPF FILE(R7399_API/MSTRKVV) SRCFILE(R7399_API/QDDSSRC) SRCMBR(MSTRKVV)  
GENLVL(20) FLAG(0) FILETYPE(*DATA) MBR(*FILE) TEXT(*SRCMBRTXT)
```

2. Command SET\_MSTR\_K:  

```
CRTCMD CMD(R7399_API/SET_MSTR_K) PGM(R7399_API/SET_MSTR_K)
VLDCKR(R7399_API/SSL_TEST) SRCFILE(R7399_API/QCMD SRC) SRCMBR(SET_MSTR_K)
```
3. Command validation program SSL\_TEST:  

```
CRTBNDC L PGM(R7399_API/SSL_TEST) SRCFILE(R7399_API/QCL SRC) SRCMBR(SSL_TEST)
```
4. ILE modules used by the command processing program:
  - a. SET\_MSTR\_K is the main command-processing program:  

```
CRTCMOD MODULE(R7399_API/SET_MSTR_K) SRCFILE(R7399_API/QC SRC)
```
  - b. BASE64\_COD converts data from/to a binary to/from Base64 format:  

```
CRTCMOD MODULE(R7399_API/BASE64_COD) SRCFILE(R7399_API/QC SRC)
```
  - c. GETTIME retrieves the system time stamp (for example, 2007-05-17-20.16.31.516000):  

```
CRTRPGMOD MODULE(R7399_API/GETTIME) SRCFILE(R7399_API/QRPG SRC)
```
  - d. RTVIN F retrieves job information:  

```
CRTCLMOD MODULE(R7399_API/RTVIN F) SRCFILE(R7399_API/QCL SRC)
```
5. Command processing program SET\_MSTR\_K:  

```
CRTPGM PGM(R7399_API/SET_MSTR_K) MODULE(R7399_API/SET_MSTR_K
R7399_API/BASE64_COD R7399_API/GETTIME R7399_API/RTVIN F)
```

**Alternative method: Create a program**

The above steps create a command, SET\_MSTR\_K. If you want to achieve the same goal in a program way, you can create a program, CRT\_MK, instead. To do this:

1. Create CL program CRT\_MK:  

```
CRTBNDC L PGM(R7399_API/CRT_MK) SRCFILE(R7399_API/QCL SRC)
SRCMBR(CRT_MK) TEXT(*SRCMBRTXT)
```
2. Start this program:  

```
CALL CRT_MK
```

**Symmetric key creation (GEN\_SYMKEY)**

To build this command, create the following objects:

1. Physical file INFOFILE:  

```
CRTPF FILE(R7399_API/INFOFILE) SRCFILE(R7399_API/QDD SRC) SRCMBR(INFOFILE)
```
2. Command GEN\_SYMKEY:  

```
CRTCMD CMD(R7399_API/GEN_SYMKEY) PGM(*LIBL/GEN_SYMKEY)
SRCFILE(R7399_API/QCMD SRC) SRCMBR(GEN_SYMKEY)
```
3. ILE modules used by the command processing program:
  - a. GEN\_SYMKEY is the main command-processing program:  

```
CRTCMOD MODULE(R7399_API/GEN_SYMKEY) SRCFILE(R7399_API/QC SRC)
```
  - b. BASE64\_COD converts data from/to a binary to/from Base64 format:  

```
CRTCMOD MODULE(R7399_API/BASE64_COD) SRCFILE(R7399_API/QC SRC)
```
  - c. GETTIME retrieves the system time stamp (for example, 2007-05-17-20.16.31.516000):  

```
CRTRPGMOD MODULE(R7399_API/GETTIME) SRCFILE(R7399_API/QRPG SRC)
```

d. RTVINP retrieves job information:

```
CRTCLMOD MODULE(R7399_API/RTVINP) SRCFILE(R7399_API/QCLSRC)
```

4. Command processing program GEN\_SYMKEY:

```
CRTPGM PGM(R7399_API/GEN_SYMKEY) MODULE(R7399_API/GEN_SYMKEY  
R7399_API/BASE64_COD R7399_API/GETTIME R7399_API/RTVINP)
```

### **Alternative method: Create a program**

The above steps create a command, GEN\_SYMKEY. If you want to achieve the same goal in a program way, you can create a program, CRT\_SK, instead. To do this:

1. Create CL program CRT\_SK:

```
CRTBNDCL PGM(R7399_API/CRT_SK) SRCFILE(R7399_API/QCLSRC)  
SRCMBR(CRT_SK) TEXT(*SRCMBRTXT)
```

2. Start this program:

```
CALL CRT_SK
```

### **Encrypt and store data (SET\_DATA)**

To build this command, create the following objects:

1. The physical file ENCDATA stores the encrypted data:

```
CRTPF FILE(R7399_API/ENCDATA) SRCFILE(R7399_API/QDDSSRC) SRCMBR(ENCDATA)  
GENLVL(20) FLAG(0) FILETYPE(*DATA)
```

2. Command SET\_DATA:

```
CRTCMD CMD(R7399_API/SET_DATA) PGM(R7399_API/SET_DATA)  
SRCFILE(R7399_API/QCMDSRC) SRCMBR(SET_DATA)
```

3. Command-processing program SET\_DATA:

```
CRTBNDCL PGM(R7399_API/SET_DATA) SRCFILE(R7399_API/QCLSRC) SRCMBR(SET_DATA)
```

4. ILE modules used by the ENC\_DATA program:

a. ENC\_DATA performs the main data encryption functions:

```
CRTCMOD MODULE(R7399_API/ENC_DATA) SRCFILE(R7399_API/QCSRC)
```

b. BASE64\_COD converts data from/to a binary to/from Base64 format:

```
CRTCMOD MODULE(R7399_API/BASE64_COD) SRCFILE(R7399_API/QCSRC)
```

c. GETTIME retrieves the system time stamp (for example, 2007-05-17-20.16.31.516000):

```
CRTPRGMOD MODULE(R7399_API/GETTIME) SRCFILE(R7399_API/QRPGSRC)
```

5. Program ENC\_DATA performs the main data encryption functions:

```
CRTPGM PGM(R7399_API/ENC_DATA) MODULE(R7399_API/ENC_DATA R7399_API/BASE64_COD  
R7399_API/GETTIME)
```

### **Alternative method: Create a program**

The above steps create a command, SET\_DATA. If you want to achieve the same goal in a program way, you can create a program, CRT\_SET, instead. To do this:

1. Create CL program CRT\_SET:

```
CRTBNDCL PGM(R7399_API/CRT_SET) SRCFILE(R7399_API/QCLSRC)  
SRCMBR(CRT_SET) TEXT(*SRCMBRTXT)
```

2. Start this program:

```
CALL CRT_SET
```

### **Retrieve encrypted data (GET\_DATA)**

To build this command, create the following objects:

1. Command GET\_DATA:

```
CRTCMD CMD(R7399_API/GET_DATA) PGM(*LIBL/GET_DATA) SRCFILE(R7399_API/QCMSRC) SRCMBR(GET_DATA)
```

2. Command processing program GET\_DATA:

```
CRTBNDCL PGM(R7399_API/GET_DATA) SRCFILE(R7399_API/QCLSRC) SRCMBR(GET_DATA)
```

3. ILE modules used by the DEC\_DATA program:

- a. DEC\_DATA performs the main data decryption function:

```
CRTCMOD MODULE(R7399_API/DEC_DATA) SRCFILE(R7399_API/QCSRC)
```

- b. BASE64\_COD converts data from/to a binary to/from Base64 format:

```
CRTCMOD MODULE(R7399_API/BASE64_COD) SRCFILE(R7399_API/QCSRC)
```

- c. GETTIME retrieves the system time stamp (for example, 2007-05-17-20.16.31.516000):

```
CRTRPGMOD MODULE(R7399_API/GETTIME) SRCFILE(R7399_API/QRPGSRC)
```

4. Program DEC\_DATA performs the main decryption functions:

```
CRTPGM PGM(R7399_API/DEC_DATA) MODULE(R7399_API/DEC_DATA R7399_API/BASE64_COD R7399_API/GETTIME)
```

### **Alternative method: Create a program**

The above steps create a command, GET\_DATA. If you want to achieve the same goal in a program way, you can create a program, CRT\_GET, instead. To do this:

1. Create CL program CRT\_GET:

```
CRTBNDCL PGM(R7399_API/CRT_GET) SRCFILE(R7399_API/QCLSRC) SRCMBR(CRT_GET) TEXT(*SRCMBRTXT)
```

2. Start this program:

```
CALL CRT_GET
```

### **Translate Key Store (TRANS\_KEY)**

To build this command, create the following objects:

1. Command TRANS\_KEY:

```
CRTCMD CMD(R7399_API/TRANS_KEY) PGM(R7399_API/TRANS_KEY)
```

2. Program TRANS\_KEY translates keys stored in the specified keystore files to another master key:

```
CRTBNDCL PGM(R7399_API/TRANS_KEY) SRCFILE(R7399_API/QCSRC) SRCMBR(TRANS_KEY)
```

### ***Iternative method: Create a program***

The above steps create a command, TRANS\_KEY. If you want to achieve the same goal in a program way, you can create a program, CRT\_TK, instead. To do this:

1. Create CL program CRT\_GET:

```
CRTBNDCL PGM(R7399_API/CRT_TK) SRCFILE(R7399_API/QCLSRC)
          SRCMBR(CRT_TK) TEXT(*SRCMBRTXT)
```

2. Start this program:

```
CALL CRT_TK
```

## **11.3 Using the scenario application**

Now that we have created the application components, we are ready to use the application. This section walks you through the entire process including:

1. Creating a masker key: SET\_MSTR\_K command
2. Creating symmetric keys: GEN\_SYMKEY command
3. Encrypting data: SET\_DATA command
4. Decrypting data on source system: GET\_DATA command
5. Translating Key Store: TRANS\_KEY command
6. Decrypting data on target system: GET\_DATA command

### **11.3.1 Create a master key: SET\_MSTR\_K command**

The first step in using our application is to establish a master key. In our scenario, perform this step by running the SET\_MSTR\_K command.

**Note:** As a reminder, a master key is a 256-bit AES key used by the Cryptographic Services APIs to protect other keys. i5/OS supports a maximum of eight master keys, which are stored in a specially protected area of the Licensed Internal Code.

For more information about master keys, along with the three master key *versions*, refer to Chapter 3, “Key management concepts” on page 25, and Chapter 5, “Managing keys on System i” on page 47.

## Executing SET\_MSTR\_K command

This command controls the creation of the master key for use in our application. Enter the following command on your i5/OS command entry screen:

```
SET_MSTR_K
```

Then press F4 for the prompt screen shown in Figure 11-6.

```

                                Set Master Key (SET_MSTR_K)

Type choices, press Enter.

Master Key ID . . . . . 8          1, 2, 3, 4, 5, 6, 7, 8
Clear Master Key Version . . . . *NEW      *NEW, *OLD, *BOTH
Passphrase Part 1 (25 Char) . .
  Reenter to confirm . . . . .
Passphrase Part 2 (25 Char) . .
  Reenter to confirm . . . . .
Passphrase Part 3 (25 Char) . .
  Reenter to confirm . . . . .
SSL Required . . . . . *YES        *YES, *NO

                                                                Bottom
F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
F24=More keys
```

Figure 11-6 The Set Master Key (SET\_MSTR\_K) CL command prompt

The command requires you to supply the following information:

► **Master Key ID (MSTRD)**

This defines which master key you wish to set. (Remember, there are eight available in i5/OS.)

As our scenario involves distributing data to a second system, or a target system, it is necessary to use a key ID that is available on both systems. This key ID will be used on the target system as a temporary master key when moving over the keystore file.

► **Clear Master Key Version (CLR)**

This defines which *version* of the master key (identified in the Master Key ID parameter) you wish to clear.

► **Passphrase Part x (25 Char) (IDx\_1)**

These are the passphrases that will be combined to generate the key.

Our command has three passphrases available to allow segregation of the key parts amongst different individuals. This ensures that no one person has all of the information necessary to recreate the master key.

Each passphrase in our command allows up to 25 characters of text.

- ▶ Reenter to confirm (IDx\_2)

The command masks the entry of the passphrases, and therefore requires dual entry of each passphrase to prevent key stroke errors.

- ▶ SSL Required (SSL)

If the user requests SSL, then the command will validate that the user is connected to an SSL-enabled 5250 session before processing the request. We recommend this in order to prevent the supplied information from being observable across the network.

Upon invocation, the command passes the user-supplied information to the command validation program (SSL\_TEST), which checks to see whether the user requested that SSL be verified. If so, it accomplishes the verification by retrieving the display device information from the current job, as shown in Figure 11-7.

```
/* RETRIEVE JOB ATTRIBUTES - JOB NAME (= DEVICE NAME)
      RTVJOBA    JOB(&DEVNAME)
/* RETRIEVE DEVICE DESCRIPTION API
      CALL      PGM(QSYS/QDCRDEVD) PARM(&DEVINFO &LENGTH +
      'DEVDO600' &DEVNAME &ERRINFO)
```

Figure 11-7 Retrieve display device information

Using the Retrieve Device Description API, we determine the TCP/IP port number that the user is connected to. Using the port number, we can verify whether the client connection to the server is secure (the default communication port for SSL Telnet is 992). If the user indicated that SSL is required, and the device is communicating over a unsecure connection, the program displays the message `No SSL connection` and returns to the command display to await user correction.

If SSL verification is passed (or not required), control passes to the command processing program (SET\_MSTR\_K) to perform the key action.

## Understanding SET\_MSTR\_K command

**Note:** This section provides a detailed and structural analysis of the SET\_MSTR\_K command. If you want to quickly run the entire application to have a complete view, you may skip to “Executing GEN\_SYMKEY command” on page 153. However, we strongly recommend that you come back to this point and read the rest of this section carefully to understand the logic behind this.

This section provides a detailed functional and structural analysis of the SET\_MSTR\_K command.

The SET\_MSTR\_K command requires the user to provide the following main information:

- ▶ Which one of the eight master keys we wish to use
- ▶ The passphrases to be used to generate the master key

The rest of the information required by the APIs is coded into the command processing program:

- ▶ How many passphrases to combine to make the master key
- ▶ Length of each passphrase
- ▶ CCSID for each passphrase

In Figure 11-8, we have selected master key 8 and that we want to clear the *new* version of the master key. We have also provided our passphrases and that an SSL connection is required.

```

                                Set Master Key (SET_MSTR_K)

Type choices, press Enter.

Master Key ID . . . . . > 8                1, 2, 3, 4, 5, 6, 7, 8
Clear Master Key Version . . . . . > *NEW    *NEW, *OLD, *BOTH
Passphrase Part 1 (25 Char) . . . . . >
  Reenter to confirm . . . . . >
Passphrase Part 2 (25 Char) . . . . . >
  Reenter to confirm . . . . . >
Passphrase Part 3 (25 Char) . . . . . >
  Reenter to confirm . . . . . >
SSL Required . . . . . > *YES                *YES, *NO
```

Figure 11-8 Using the SET\_MSTR\_K CL command to create master key 8

**Note:** Before clearing an old master key version, care should be taken to ensure that no key encryption keys or data encryption keys are still encrypted under the old version of the master key.

### Used APIs

The SET\_MSTR\_K command (setting up a master key, in other words) utilizes the APIs reviewed in this section.

#### **Clear Master Key API**

This API is *Qc3ClearMasterKey* in ILE and *QC3CLRMK* in OPM.

The first step is to clear the new master key version. The call to the Clear Master Key API is depicted in Example 11-1.

Example 11-1 Clear Master Key API

---

```
memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
Qc3ClearMasterKey ( &mkid,
                   &mstk,
                   &errCode);
if ( errCode.Bytes_Available != 0 ) {
    printf (">>Qc3ClearMasterKey< \n");
    printf ("errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}
```

---



The Clear Master Key API clears a *new* or *old* master key version by setting it to binary 0s. The parameter list includes:

► **mkid**

The master key ID indicates which of the eight key positions is going to be used. We chose position 8.

► **mstkv**

This is the master key version to be cleared. We are creating a new key (hence our \*NEW designation on the SET\_MSTR\_K command). The value of hex "F0" indicates the new key version.

► **errCode**

Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of the Clear Master Key API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/apis/qc3clrmk.htm>

### **Load Master Key Part API**

This API is *Qc3LoadMasterKeyPart* in ILE and *QC3LDMKP* in OPM.

This API loads one or more parts that will be used to generate the master key. Although our command allows for three passphrases, this is a self-imposed limitation and is not representative of the number of parts allowed by the master key APIs. Still, it is recommended to assign different key parts to different individuals to ensure that no single person has the ability to reproduce a master key.

We call the Load Master Key Part API for each of the three passphrases entered on the SET\_MSTR\_K CL command. The call to the Load Master Key Part API for the first passphrase is depicted in Example 11-2.

*Example 11-2 Load Master Key Part API*

---

```
CCSID_of_passphrase = 37;
Length_of_passphrase = sizeof(Passphrase_1);
memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
Qc3LoadMasterKeyPart ( &mkid,
                      &Passphrase_1,
                      &Length_of_passphrase,
                      &CCSID_of_passphrase,
                      &errCode );
if ( errCode.Bytes_Available != 0 ) {
    printf (">>Qc3LoadMasterKeyPart<< \n");
    printf ("errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}
```

---

The parameter list includes:

- ▶ **mkid**  
The master key ID indicates which of the eight key positions is going to be used. We chose position 8.
- ▶ **Passphrase\_1**  
This is the first passphrase selected on the SET\_MSTR\_K command (this API is called once for each passphrase).
- ▶ **Length\_of\_passphrase**  
This is the length of the passphrase. The SET\_MSTR\_K command allows for the entry of passphrases up to 25 characters in length.
- ▶ **CCSID\_of\_passphrase**  
We are using a CCSID of 37.
- ▶ **errCode**  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:  
<http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp>  
Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/apis/qc3ldmkp.htm>

### **Set Master Key API**

This API is *Qc3SetMasterKey* in ILE and *QC3SETMK* in OPM.

Once all of the master key parts are loaded, we need to set the master key to finalize it. During the set function, the generated key (and its Key Verification Value KVV) is moved into the master key's *current* version. The existing current version is moved to the *old* version, and the existing old version is dropped.

The Set Master Key API sets the master key versions as follows:

- ▶ Moves the *current* master key version into the *old* master key version
- ▶ Moves the *new* master key version into the *current* master key version
- ▶ Clears the *new* master key version by settings it to binary 0s

The call to the Set Master Key API is depicted in Example 11-3.

#### *Example 11-3 Set master key API*

---

```
memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
Qc3SetMasterKey    ( &mkid,
                    &Key_verification_value,
                    &errCode);
if ( errCode.Bytes_Available != 0 ) {
    printf (">>Qc3SetMasterKey< \n");
    printf ("errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}
```

---

The parameter list includes:

► **mkid**

The master key ID indicates which of the eight key positions is going to be used. We chose position 8.

► **Key\_verification\_value**

The key verification value (KVV) is a 20-character value returned from the API. The KVV can be used for later reference to determine whether the master key has changed.

The *current* and *old* master key versions each have a KVV.

When you set a master key, any key encrypting key or data encryption key that was stored under that master key must be translated (decrypted and reencrypted). For more information, see 5.1.3, “Changing a master key” on page 52.

► **errCode**

Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp?topic=/apis/qc3setmk.htm>

### **Calculate Hash API**

This API is *Qc3CalculateHash* in ILE and *QC3CALHA* in OPM. The Calculate Hash API calculates a hash value of a string.

Although not a requirement of creating a master key, we decided to store the key verification value returned by the set master key API for later reference. However, as we are storing the KVV in a regular database file, we first hash the value using the SHA-1 hashing algorithm. We then convert the result to base64 to ensure that it only contains displayable characters.

The call to the Calculate Hash API is depicted in Example 11-4.

#### *Example 11-4 Qc3CalculateHash API*

---

```
memset (&Key_verification_Hash, '\0', sizeof(Key_verification_Hash));
Length_of_input_data = sizeof(Key_verification_value);
Algorithm_description[3] = Qc3_SHA1;
Cryptographic_service_provider = Qc3_Any_CSP;

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
Qc3CalculateHash (&Key_verification_value,
                  &Length_of_input_data,
                  "DATA0100",
                  &Algorithm_description,
                  "ALGD0500",
                  &Cryptographic_service_provider,
                  NULL,
                  &Key_verification_Hash,
                  &errCode );
```

```

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3CalculateHash< \n");
    printf ("errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}

```

---

The parameter list includes:

- ▶ **Key\_verification\_value**  
This is the KVV provided by the Set Master Key API.
- ▶ **Length\_of\_input\_data**  
This is the length of the KVV.
- ▶ **"DATA0100"**  
This indicates to the API the format of the input data. This format indicates that the `key_verification_value` parameter contains the data to be hashed, rather than an array of pointers.
- ▶ **Algorithm\_description**  
For SHA-1, this will be binary 00000002.
- ▶ **"ALGD0500"**  
This indicates to the API which format we wish to use for the `Algorithm_description` parameter. ALGD0500 specifies the structure for a hash algorithm.
- ▶ **Cryptographic\_service\_provider**  
This indicates which cryptographic service provider (CSP) will perform the hash operation—software or hardware. Using a character value of 0 instructs i5/OS to select the appropriate CSP.
- ▶ **Key\_verification\_Hash**  
This is the area in which to store the returned hash value. The length of the hash is defined by the hash algorithm. For SHA-1 this will be a character value of 20 bytes.
- ▶ **errCode**  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp?topic=/apis/qc3calha.htm>

## Record generated master key details

To store information about the generated master keys, we write the following details to the file MSTRKVV:

- ▶ Master key number
  - This is the master key ID that was created (this field is used as the database key field).
- ▶ Key verification value
  - This is the KVV returned by the Set Master Key API, which was subsequently hashed using SHA-1 and converted to Base64.
- ▶ Time stamp of when the master key was created
  - We retrieve the current system time using RPG routine GETTIME.
- ▶ User profile under which the master key was generated
  - We retrieve the user profile name in CL module RTVINP.
- ▶ System name where the master key was generated
  - We retrieve the system name in CL module RTVINP.

To see information about the current master keys, enter the following CL command:

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/MSTRKVV)
```

Sample results are depicted in Figure 11-9.

MSTRNUM	KVV	TIME	USER
8	pqUBIqLfyThueZ+QmeEGvLIOWUO=	2007-05-18-00.21.28.209000	MILANK
1	0XkNLzhc7PgzoRdMCPJsCj0o1+8=	2007-05-18-00.30.25.279000	MILANK
5	pqUBIqLfyThueZ+QmeEGvLIOWUO=	2007-05-18-00.30.41.549000	MILANK

Figure 11-9 Sample master key information

**Note:** We assume that master keys will only be set up via the SET\_MSTR\_K command. Although it is useful to know which master keys are already established on the system, the KVV stored in database file MSTRKVV is not the true KVV. The KVV generated by the Set Master Key API is first hashed using the SHA-1 algorithm, and then converted to Base64 for readability.

## Running SET\_MSTR\_K command step summary

The basic steps that we took to generate our master key are shown in Figure 11-10.

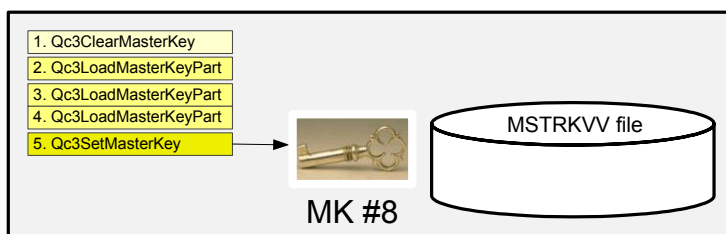


Figure 11-10 Master key and MSTRKVV file

### 11.3.2 Create symmetric keys: GEN\_SYMKEY command

The second step is to generate two symmetric keys: one for the source system and one for the target system.

**Note:** A symmetric key is a single key that can be used to encrypt, and also subsequently decrypt, data.

In i5/OS, a symmetric key is stored in a database file called a keystore file. The keys in the keystore file are encrypted under a master key.

#### ***Disallowed function***

When a key is created, it can be limited to only perform certain cryptographic functions. This is controlled via the *disallowed function* parameter, and the options are listed below:

- ▶ No functions are disallowed (binary 0).
- ▶ Encryption is disallowed (binary 1).
- ▶ Decryption is disallowed (binary 2).
- ▶ MACing is disallowed (binary 4).
- ▶ Signing is disallowed (binary 8).

The values can be summed in order to disallow multiple functions. For example, to disallow everything *except* MACing, the sum of the values is  $1+2+8 = 11$  (Hex B). This value should be saved along with the encrypted key value because it will be required when the encrypted key value is used on the Cryptographic Services APIs.

## Executing GEN\_SYMKEY command

This command controls the creation of the symmetric keys for use in our application, and appears as shown in Figure 11-11.

GEN\_SYMKEY

```

                                Generate AES Keys (GEN_SYMKEY)

Type choices, press Enter.

Key LABEL of Key record (KL) . . LABEL_04
Master Key (MK) . . . . . 8          1, 2, 3, 4, 5, 6, 7, 8
Key Store File Source (KSFS) . . KEYFILES  Name
  Library Name . . . . . R7399_API  Name, *CURLIB
TEXT . . . . . SOURCE          Character value
Key Store File Target (KSFT) . . KEYFILET  Name
  Library Name . . . . . R7399_API  Name, *CURLIB
TEXT . . . . . TARGET          Character value
Allowed Function Target (DF) . . D          E, D, S, M, A
Info File (IF) . . . . . INFOFILE  Name
  Library Name . . . . . R7399_API  Name, *CURLIB
KEK file (KEKF) . . . . . KEKFILE  Name
  Library Name . . . . . R7399_API  Name, *CURLIB

                                                                Bottom
F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this display
F24=More keys

```

Figure 11-11 CMD to generate symmetric keys prompt

The parameters are:

- ▶ Key LABEL of Key record (KEYLABEL)
 

This is a text label used to identify the keys in the keystore.
- ▶ Master key (MSTRK)
 

This is the master key ID of the master key that will be used to encrypt the keys in the keystore files.
- ▶ Key Store File Source (KEYFILES)
 

This is the qualified name of the keystore file to store the key on the *source system*.
- ▶ Text (TEXT1)
 

Text description to be stored in the information file (see INFOFILE parameter) on the *Source system*.
- ▶ Key Store File Target (KEYFILET)
 

This is the qualified name of the keystore file to store the key on the *target system*. This file will need to be moved to the target system after creation.
- ▶ Text (TEXT2)
 

Text description to be stored in the information file (see INFOFILE parameter) on the target system.

► Allowed Function Target (DISALL)

This code indicates which key functions are allowed for the key created in the Key Store File Target keystore:

<b>E</b>	Encryption is allowed.
<b>D</b>	Decryption is allowed.
<b>S</b>	Signing is allowed.
<b>M</b>	MACing is allowed.
<b>A</b>	All of the above are allowed.

In our scenario, we pick the value D so that the only decryption function is to be allowed on the target system.

► Info File (INFOFILE)

This is the qualified name of the file used to store information about the generated keys.

► KEK File (KEKFILE)

This is the qualified name of the Key Encryption Key (KEK) keystore file.

## Understanding the GEN\_SYMKEY command

**Note:** This section provides a detailed and structural analysis of the GEN\_SYMKEY command. If you want to quickly run the entire application to have a complete view, you may skip to “Executing SET\_DATA command” on page 171. However, we strongly recommend that you come back to this point and read the rest of section carefully to understand the logic behind this.

This section provides some background information about the GEN\_SYMKEY command.

### *Use of symmetric keys for source and target systems*

As already mentioned, we need to create two symmetric keys for our application. One key will be dedicated to source system and will allow both encryption and decryption functions. The other key is intended for use on target system and allows only decryption functions.



We accomplish this by setting up two separate keystore files. The same key will be written to both keystore files with the same label. This is depicted in Figure 11-12. The only difference between the key records is the disallowed functions:

- ▶ Source system
 

The key intended for use on the source system will have a disallow function setting of '0', which means that no function is disallowed (that is, it is capable of performing all functions).
- ▶ Target system
 

The key intended for use on the target system will have a disallow function setting of Hex D (13), which indicates that all functions except decryption are disallowed (that is, it can *only* perform decryption).

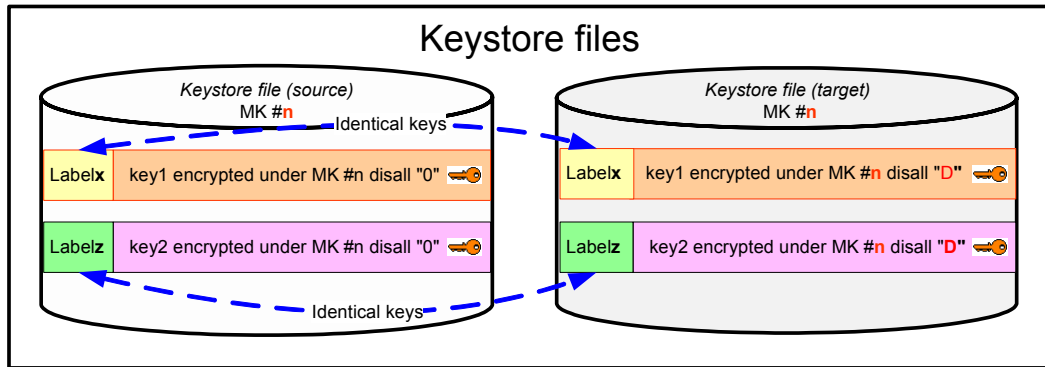


Figure 11-12 Keystore files

### Setting up the symmetric keys

Our GEN\_SYMKEY command requires that the user provide the following main information:

- ▶ The key label to be assigned to the two keys for later retrieval
- ▶ Which of the eight master keys should be used to encrypt the keys

In Figure 11-13, we have selected LABEL\_04 as the desired key label, and master key 8. This is the master key that we created in “Executing SET\_MSTR\_K command” on page 144.

The other parameters are left at their default values.

```

Generate AES Keys (GEN_SYMKEY)

Type choices, press Enter.

Key LABEL of Key record (KL) . . LABEL_04
Master Key (MK) . . . . . 8          1, 2, 3, 4, 5, 6, 7, 8
  
```

Figure 11-13 Using the GEN\_SYMKEY CL command to specify which label and master key to use

### Used APIs

To set up a symmetric key, we utilize the following APIs.

#### Create Keystore API: for source system

This API is *Qc3CreateKeyStore* in ILE and *QC3CRTKS* in OPM.

The Create Key Store API generates a keystore file in a library. The key values in the keystore file are encrypted under a given master key. We use this API several times to generate the three keystores (a keystore file for the source and target system and KEK keystore file).

In preparation for generating the symmetric key, we first need to create the keystore file for the source system. The call to the Create Keystore API is depicted in Example 11-5.

*Example 11-5 Create Keystore API*

---

```

*++argv;
memset( kskey_S.Key_Store, ' ', sizeof(kskey_S.Key_Store));
memcpy( kskey_S.Key_Store, argv[0], 20);
memcpy( ksauth, "*EXCLUDE ", 10);
memset( ObjExist, 0, sizeof(ObjExist));
memcpy( ObjExist, "/QSYS.LIB/", 10);

for ( i=10,j=10; i < 20 && kskey_S.Key_Store[i] != ' ';
      ObjExist[j] = kskey_S.Key_Store[i], ++i, ++j);
      memcpy(&ObjExist[j], ".LIB/", 5);
      j += 5;
for ( i= 0      ; i < 10 && kskey_S.Key_Store[i] != ' ';
      ObjExist[j] = kskey_S.Key_Store[i], ++i, ++j);
      memcpy(&ObjExist[j], ".FILE", 5);

if ( stat( ObjExist, &statbuf ) != 0 )
{
    memset( ksdesc, ' ', sizeof(ksdesc));
    memcpy( ksdesc, ">>> KSF <S< ", 13 );
    memcpy(&ksdesc[13], argv[0], 10);
    memset(&errCode, 0, sizeof(errCode));
    errCode.Bytes_Provided = sizeof(errCode);

    Qc3CreateKeyStore( kskey_S.Key_Store,
                       &mkid,
                       ksauth,
                       ksdesc,
                       &errCode);

    if ( errCode.Bytes_Available != 0 )
    {
        printf (">>Qc3CreateKeyStore < \n");
        printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
        return;
    }
}

```

---

**Note:** If the keystore already exists, then any keys stored there will be encrypted by the master key specified during the keystore creation and *not* under the master key requested on the GEN\_SYMKEY command.

The parameter list includes:

► kskey\_S.Key\_Store

This is the qualified name of the desired keystore file. For us, this will be KEYFILES in library R7399\_API.

- ▶ **mkid**  
This is the master key that the keystore will be encrypted under. We use master key 8.
- ▶ **ksauth**  
This is the i5/OS object-level authority that \*PUBLIC will have to the keystore file. The recommended value is \*EXCLUDE.
- ▶ **ksdesc**  
This is the text description that will be given to the keystore file object.
- ▶ **errCode**  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/apis/qc3crtks.htm>

### **Create Keystore API: for target system**

This API is *Qc3CreateKeyStore* in ILE and *QC3CRTKS* in OPM. This is the same API used to create a keystore file on the source system.

We repeat the creation for the keystore file for the target system. Again, we use the Create Keystore API.

The parameter list includes:

- ▶ **kskey\_T.Key\_Store**  
This is the qualified name of the desired keystore file. For us, this will be KEYFILET in library R7399\_API.
- ▶ **mkid**  
This is the master key that the keystore will be encrypted under. We use master key 8.
- ▶ **ksauth**  
This is the i5/OS object-level authority that \*PUBLIC will have to the keystore file. The recommended value is \*EXCLUDE.
- ▶ **ksdesc**  
This is the text description that will be given to the keystore file object.
- ▶ **errCode**  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

**Note:** Again, if the keystore already exists, then any keys stored there will be encrypted by the master key specified during the keystore creation and *not* under the master key requested on the GEN\_SYMKEY command.

### **Create Keystore API: for key encryption key**

This API is *Qc3CreateKeyStore* in ILE and *QC3CRTKS* in OPM. This is the same API used to create a keystore file on both source and target systems.

Now we create the final keystore file. This keystore file is to contain a key encrypting key (KEK). Although the KEK is not actually *required* (as the data encryption key can be encrypted directly under the master key), we use a KEK to ensure that the data encryption key is not maintained in the program in cleartext form.

As before, we use the Create Keystore API.

The parameter list includes:

▶ **kskey\_K.Key\_Store**

This is the qualified name of the desired keystore file. For us, this will be KEKFILE in library R7399\_API.

▶ **mkid**

This is the master key that the keystore will be encrypted under. We use master key 8.

▶ **ksauth**

This is the i5/OS object-level authority that \*PUBLIC will have to the keystore file. The recommended value is \*EXCLUDE.

▶ **ksdesc**

This is the text description that will be given to the keystore file object.

▶ **errCode**

Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

### **Generate Key Record API**

This API is *Qc3GenKeyRecord* in ILE and *QC3GENKR* in OPM.

The Generate Key Record API generates a random key and stores it in a keystore file. We use this API to generate and store a key encrypting key.

This API generates a 256-bit AES key encrypting key and stores it in the KEK keystore file. The call to the Generate Key Record API is depicted in Example 11-6.

*Example 11-6 Qc3GenKeyRecord for KEK*

```
memset(kskey_K.Record_Label, 0x40, sizeof(kskey_K.Record_Label));
memcpy(kskey_K.Record_Label, "KEK_LABEL_01", 12 );
keyType = Qc3_AES;
keySize = 32;
pubExp = 0;
```

```

disFunc=0;
memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

Qc3GenKeyRecord( kskey_K.Key_Store,
                  kskey_K.Record_Label,
                  &keyType,
                  &keySize,
                  &pubExp,
                  &disFunc,
                  &Cryptographic_service_provider,
                  NULL,
                  &errCode);

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3GenKeyRecord < \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}

```

---

The parameter list includes:

- ▶ **kskey\_K.Key\_Store**  
This is the qualified name of the KEK keystore file. For us, this will be KEKFILE in library R7399\_API.
- ▶ **kskey\_K.Record\_Label**  
Record label. This is hardcoded in the program as “KEK\_LABEL\_01”. This enables retrieval of the correct key.
- ▶ **keyType**  
This is the key type. We are generating an AES key, so we designate “Qc3\_AES”.
- ▶ **keySize**  
This is the size of the key that will be generated. For a 256-bit key, this value is 32 (32 x 8 = 256).
- ▶ **pubExp**  
The public key exponent for the symmetric key must be set to binary 0.
- ▶ **disFunc**  
The disallowed function for KEK is binary 0 (allow all functions).
- ▶ **Cryptographic\_service\_provider**  
This indicates which cryptographic service provider (CSP) will perform the hash operation—software or hardware. Using a character value of 0 instructs i5/OS to select the appropriate CSP.
- ▶ **errCode**  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/series/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iseriis/v5r4/index.jsp?topic=/apis/qc3genkr.htm>

### **Create Key Context API**

This API is *Qc3CreateKeyContext* in ILE and *QC3CRTKX* in OPM.

The Create Key Context API sets up a token that can be used to reference a key across multiple API calls.

This API creates a temporary area to be used for holding a cryptographic key, and returns a token that can be used on subsequent API calls. The key context cannot be shared between jobs. The call to the Create Key Context API is depicted in Example 11-7.

**Note:** The key context includes an authentication value. If the token is used for subsequent API calls, but with an incorrect authentication value, the user will be subjected to a 10-second penalty wait. For each authentication error in the job, the penalty wait increases in 10-second increments up to a maximum wait of 10 minutes.

#### *Example 11-7 Qc3CreateKeyContext API*

---

```
keyStringLen = sizeof(kskey_K);    /* Set length of key string */
keyFormat = Qc3_KSLabel_Struct;    /* Key format is keystore label*/
keyForm = Qc3_Clear;              /* Key string is clear */
                                   /* Key type already set to AES */
                                   /* Create key context */
memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

Qc3CreateKeyContext((char*)&kskey_K,
                   &keyStringLen,
                   &keyFormat,
                   &keyType,
                   &keyForm,
                   NULL,
                   NULL,
                   KEKctx,
                   &errCode);

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3CreateKeyContext <\n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}
```

---

The parameter list includes:

► **kskey\_K**

This is the key string. The key string may contain the key value or it may contain a reference to the key value.

- ▶ `keyStringLen`  
This is the length of the supplied key string.
- ▶ `keyFormat`  
This indicates the format of the supplied key string. In our case, it is a structure containing the keystore name and the record label.
- ▶ `keyType`  
This indicates the type of key. We are using binary 22, which indicates an AES key.
- ▶ `keyForm`  
The key form indicates whether the supplied key string is in an encrypted state.
- ▶ `KEKctx`  
This is the area in which the API will store the created key context token.
- ▶ `errCode`  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:  
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp>  
Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/apis/qc3crtkx.htm>

### **Create Algorithm Context API**

This API is `Qc3CreateAlgorithmContext` in ILE and `QC3CRTAX` in OPM.

The Create Algorithm Context API sets up a token that can be used to reference an encryption algorithm and its properties and maintain the state of the operation across multiple calls.

This API creates a temporary area to be used for holding the algorithm parameters, and returns a token that can be used on subsequent API calls. The algorithm context cannot be shared between jobs. The call to the Create Algorithm Context API is depicted in Example 11-8.

**Note:** The algorithm context includes an authentication value. If the token is used for subsequent API calls, but with an incorrect authentication value, the user will be subjected to a 10-second penalty wait. For each authentication error in the job, the penalty wait increases in 10-second increments up to a maximum wait of 10 minutes.

#### *Example 11-8 Qc3CreateAlgorithmContext API*

```
memset(&algD, 0, sizeof(algD));      /* Init alg description to null*/
algD.Block_Cipher_Algo = Qc3_AES;    /* Set AES algorithm          */
algD.Block_Length = 16;              /* Block size is 16          */
algD.Mode = Qc3_CBC;                 /* Use cipher block chaining */
algD.Pad_Option = Qc3_No_Pad;        /* Do not pad                 */
                                     /* Create algorithm context   */

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
```

```

Qc3CreateAlgorithmContext( (unsigned char*)&algD,
                           Qc3_Alg_Block_Cipher,
                           AESctx,
                           &errCode);

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3CreateAlgorithmContext < \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}

```

---

The parameter list includes:

- ▶ algD  
This is the algorithm description.
- ▶ Qc3\_Alg\_Block\_Cipher  
This is the algorithm description format name. We use "ALGD0200", which is used for a block cipher algorithm (DES, triple DES, AES, and RC2).
- ▶ AESctx  
This is the area in which the API will store the created algorithm context token.
- ▶ errCode  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:  
<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp>  
Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp?topic=/apis/qc3crtax.htm>

### **Generate Symmetric Key API**

This API is *Qc3GenSymmetricKey* in ILE and *QC3GENSK* in OPM.

The Generate Symmetric Key API creates a data encryption key. In our example, we generate a 256-bit AES key.

This API generates a random key value. This key is encrypted under the KEK that we created (labeled KEK\_LABEL\_01). The call to the Generate Symmetric Key API is depicted in Example 11-9.

#### *Example 11-9 Qc3GenSymmetricKey API*

---

```

keyFormat = Qc3_Bin_String;          /* Return a binary string */
keyForm = Qc3_Encrypted;            /* Encrypt generated key */
keySize = 32;                       /* Key size is 32 bytes long */
                                     /* Key type already set to AES */
keySizeArea = 64;                   /* Receiver 64 bytes */
memset(&keyAES, 0, sizeof(keyAES));
memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

```



```

Qc3GenSymmetricKey(&keyType,
                    &keySize,
                    &keyFormat,
                    &keyForm,
                    &KEKctx,
                    &AESctx,
                    &Cryptographic_service_provider,
                    NULL,
                    keyAES,
                    &keySizeArea,
                    &keySize,
                    &errCode);

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3GenSymmetricKey < \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}

```

---

The parameter list includes:

- ▶ **keyType**  
This is the algorithm type of the key to be generated.
- ▶ **keySize**  
This is the desired size of the key.
- ▶ **keyFormat**  
This indicates the format of the output key string.
- ▶ **keyForm**  
The key form indicates whether the key string is encrypted.
- ▶ **KEKctx**  
This is the key context token for the KEK.
- ▶ **AESctx**  
This is the algorithm context token for the KEK.
- ▶ **Cryptographic\_service\_provider**  
This indicates which cryptographic service provider will perform the hash operation—software or hardware. Using a character value of 0 instructs i5/OS to select the appropriate CSP.
- ▶ **keyAES**  
This is the area in which to hold the generated key string.
- ▶ **keySizeArea**  
This is the length of the area the we provide for the key string.
- ▶ **keySize**  
This is the length of the key string returned.

► `errCode`

Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/index.jsp?topic=/apis/qc3gensk.htm>

**Write Key Record API: for source system**

This API is *Qc3WriteKeyRecord* in ILE and *QC3WRTKR* in OPM.

Next, we need to store the symmetric key in the keystore file for the source system (KEYFILES). This key is stored with a disallowed function setting of 0 (All is allowed). The Write Key Record API stores a given key value in a keystore file.

To store the generated key in the keystore file, we use the Write Key Record API, as depicted in Example 11-10.

*Example 11-10 Qc3WriteKeyRecord API for source system*

---

```
memset(kskey_S.Record_Label, 0x40, sizeof(kskey_K.Record_Label));
memcpy(kskey_S.Record_Label, infoout.KEYLABEL, sizeof(infoout.KEYLABEL));
```

```
memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
```

```
Qc3WriteKeyRecord( kskey_S.Key_Store,
                   kskey_S.Record_Label,
                   keyAES,
                   &keySize,
                   &keyFormat,
                   &keyType,
                   &disFunc_rtv_S,
                   &keyForm,
                   &KEKctx,
                   &AESctx,
                   &errCode);
```

```
if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3WriteKeyRecord< \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}
```

---

The parameter list includes:

► `kskey_S.Key_Store`

This is the qualified keystore file name for the source system.

- ▶ kskey\_S.Record\_Label  
This is the key's label.
- ▶ keyAES  
This is the actual key string.
- ▶ keySize  
This is the length of the supplied key string.
- ▶ keyFormat  
This indicates the format of the supplied key string.
- ▶ keyType  
This is the algorithm type for the key.
- ▶ disFunc\_rtv\_S  
This is the disallowed function code.
- ▶ keyForm  
The key form indicates whether the supplied key string is in an encrypted state.
- ▶ KEKctx  
This is the key context token for the KEK.
- ▶ AESctx  
This is the algorithm context token for the KEK.
- ▶ errCode  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:  
<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp>  
Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/apis/qc3wrtr.htm>

### ***Write Key Record API: for target system***

This API is *Qc3WriteKeyRecord* in ILE and *QC3WRTR* in OPM. This is the same API used for the source system.

We also need to store the same symmetric key in the keystore file for the target system (KEYFILET). However, this key is stored with a disallowed function setting to only allow decryption. The call to the Write Key Record API is depicted in Example 11-11.

#### *Example 11-11 Qc3WriteKeyRecord API for target system*

---

```
memset(kskey_T.Record_Label, 0x40, sizeof(kskey_K.Record_Label));
memcpy(kskey_T.Record_Label, infoout.KEYLABEL, sizeof(infoout.KEYLABEL));

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

Qc3WriteKeyRecord( kskey_T.Key_Store,
                  kskey_T.Record_Label,
```

```

        keyAES,
        &keySize,
        &keyFormat,
        &keyType,
        &disFunc_rtv_T,
        &keyForm,
        &KEKctx,
        &AESctx,
        &errCode);
if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3WriteKeyRecord< \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}

```

---

The parameter list includes:

- ▶ kskey\_T.Key\_Store  
This is the qualified keystore file name for the target system.
- ▶ kskey\_T.Record\_Label  
This is the key's label.
- ▶ keyAES  
This is the actual key string.
- ▶ keySize  
This is the length of the supplied key string.
- ▶ keyFormat  
This indicates the format of the supplied key string.
- ▶ keyType  
This is the algorithm type of the key being written.
- ▶ disFunc\_rtv\_T  
This is the disallowed function code. On the target system, this is hexadecimal D, which indicates that the key can only be used for decryption functions.
- ▶ keyForm  
The key form indicates whether the supplied key string is in an encrypted state.
- ▶ KEKctx  
This is the key context token for the KEK.
- ▶ AESctx  
This is the algorithm context token for the KEK.
- ▶ errCode  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

### ***Destroy Key Context API***

This API is *Qc3DestroyKeyContext* in ILE and *QC3DESKX* in OPM.

This API destroys the key context. Although a key context is destroyed when the job ends, we recommend destroying it manually to ensure that it cannot be used by any other user of the job. The call to the Destroy Key Context API is depicted in Example 11-12.

*Example 11-12 Qc3DestroyKeyContext API*

---

```
Qc3DestroyKeyContext( KEKctx,  
                      &errCode);
```

---

The parameter list includes:

▶ KEKctx

This is the key context token for the KEK.

▶ errCode

Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp?topic=/apis/qc3deskx.htm>

### ***Destroy Algorithm Context API***

This API is *Qc3DestroyAlgorithmContext* in ILE and *QC3DESAX* in OPM.

This API destroys the algorithm context. Although an algorithm context is destroyed when the job ends, we recommend destroying it manually to ensure that it cannot be used by any other user of the job. The call to the Destroy Algorithm Context API is depicted in Example 11-13.

*Example 11-13 Qc3DestroyAlgorithmContext*

---

```
Qc3DestroyAlgorithmContext( AESctx,  
                             &errCode);
```

---

The parameter list includes:

▶ AESctx

This is the algorithm context token for the KEK.

▶ errCode

Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp?topic=/apis/qc3desax.htm>

### **Clear encryption key field**

This step simply clears the key from memory. This is depicted in Example 11-14.

*Example 11-14 Wipe out encrypted file key*

---

```
memset ( keyAES, '\0', sizeof(keyAES));
```

---

### **Delete Key Record API**

This API is *Qc3DeleteKeyRecord* in ILE and *QC3DLTKR* in OPM.

This API deletes a key record from a keystore file. At this time, we wish to delete the KEK key record from the KEK keystore file, as it is no longer needed. The call to the Delete Key Record API is depicted in Example 11-15.

*Example 11-15 Qc3DeleteKeyRecord API*

---

```
Qc3DeleteKeyRecord( kskey_K.Key_Store,  
                    kskey_K.Record_Label,  
                    &errCode);  
  
if ( errCode.Bytes_Available != 0 )  
{  
    printf (">>Qc3DeleteKeyRecord < \n");  
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);  
    return;  
}
```

---

The parameter list includes:

- ▶ **kskey\_K.Key\_Store**  
This is the qualified name of the KEK keystore file.
- ▶ **kskey\_K.Record\_Label**  
This is the label of the key to be deleted. For us, this is KEK\_LABEL\_01.
- ▶ **errCode**  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp?topic=/apis/qc3dltkr.htm>

## Storing symmetric key information

To store information about the generated data keys, we write the following details to the file INFOFILE. This information will be used by the program TEST\_KEY to check for the existence of this key before data is encrypted.

- ▶ Key label  
This is the key label that is used to access the key.
- ▶ Key type  
This is the algorithm type of key that is stored.
- ▶ Key Size  
This is the byte size of the key that is stored.
- ▶ Master Key  
This is the master key ID that is used to encrypt the key.
- ▶ Disallowed function  
This is the binary value of the function that is disallowed for the key.
- ▶ Key verification value  
This is the KVV of the master key, as returned by the Set Master Key API, which was subsequently hashed using SHA-1 and converted to Base64.

To see information about the current symmetric keys, enter the following CL command:

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/INFOFILE)
```

Sample results are depicted in Figure 11-14.

KEYLABEL	KEYTYPE	KEYSIZE	SMK	SDISALL	SKVV
LABEL_07	AES-CBC	32	8	0	pqUBIqLfyThueZ+QmeEGvLIOWU0=
LABEL_01	AES-CBC	32	8	0	pqUBIqLfyThueZ+QmeEGvLIOWU0=
LABEL_04	AES-CBC	32	8	0	pqUBIqLfyThueZ+QmeEGvLIOWU0=

Figure 11-14 Sample INFOFILE file contents

## Running GEN\_SYMKEY command step summary

So far, we have created our master key, the various keystore files, a key encrypting key, and two copies of a symmetric key. The symmetric key is stored in two keystore files using the same key label, however, each key has a different disallowed function setting based on where the key will be used.

The steps are outlined in Figure 11-15.

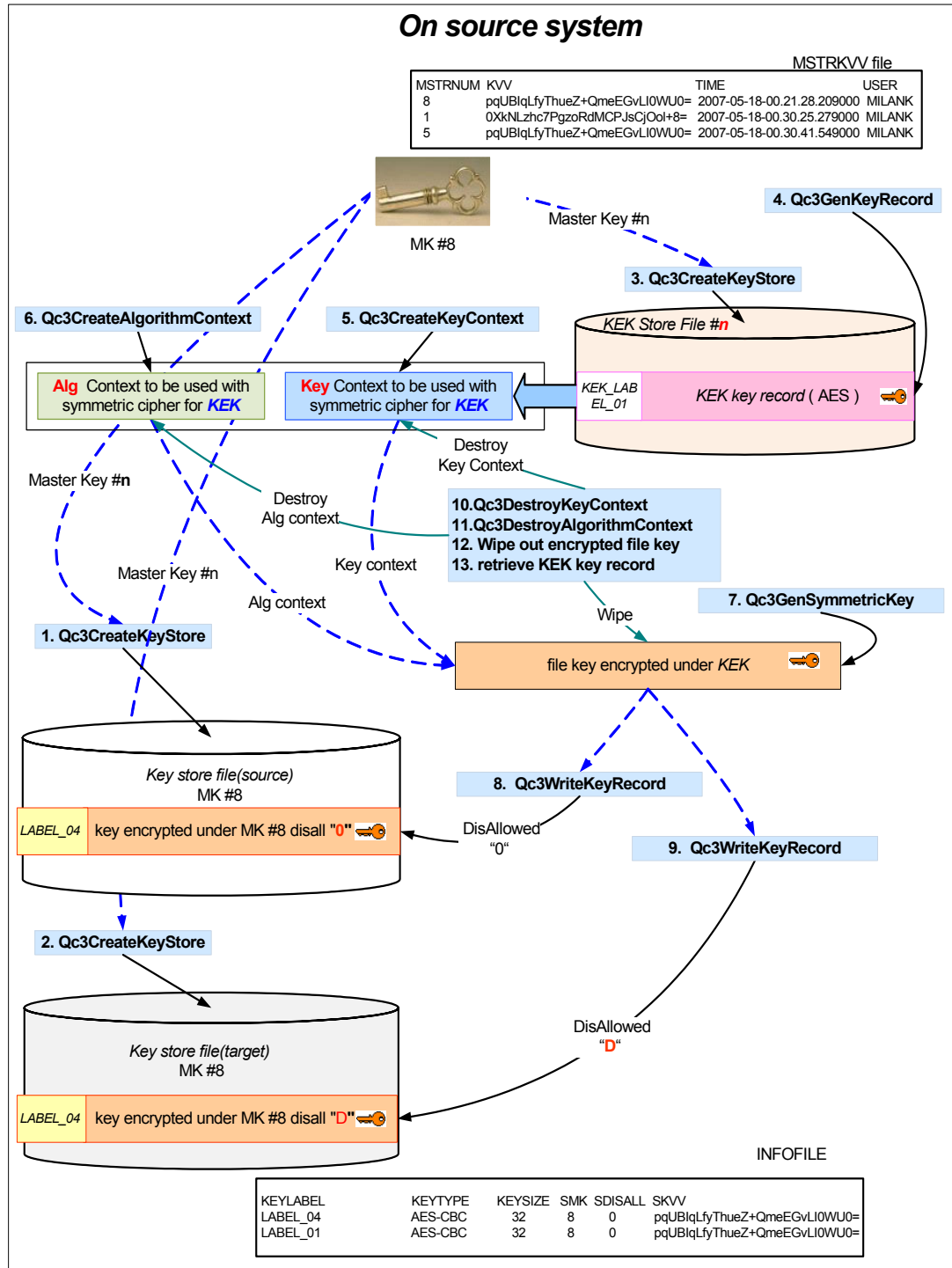


Figure 11-15 Overview of the GEN\_SYMKEY function

### 11.3.3 Encrypt data: SET\_DATA command

Now that we have our infrastructure in place, we are ready to encrypt data. We do this by running SET\_DATA command.



## Executing SET\_DATA command

This command writes data (including private information) to the database file ENCDATA on the source system.

Although this type of function is more commonly performed via an application program with a display format, we created a command-based interface to allow us to focus on the cryptographic functionality (Figure 11-16).

```
Set Customer Data (SET_DATA)

Type choices, press Enter.

Key LABEL of Key record (KL) . .
Key Store File Souce (KSFS) . . KEYFILES      Name
  Library Name . . . . . R7399_API  Name, *CURLIB
Key Store File Target (KSFT) . . KEYFILET     Name
  Library Name . . . . . R7399_API  Name, *CURLIB
Data File (DF) . . . . . ENCDATA      Name
  Library Name . . . . . R7399_API  Name, *CURLIB
First Name . . . . .
Last Name . . . . .
Address . . . . .
City . . . . .
Country . . . . .
States . . . . . Character value
ZIP Code . . . . . Character value
Phone Number . . . . . Character value
Secure Data . . . . .
```

Figure 11-16 The Set Customer Data (SET\_DATA) CL command prompt

The command requires you to supply the following information:

- ▶ **Key LABEL of Key record**  
This is the value of the label field in the keystore file indicating which key we wish to use.
- ▶ **Key Store File Source**  
This is the name of the keystore file for the source system.
- ▶ **Key Store File Target**  
This is the name of the keystore file for the target system.
- ▶ **Data File**  
This is the name of the file that contains the customer data.
- ▶ **Miscellaneous Customer Details**  
Specify the customer details (such as name and address) that you wish to add to the customer data file.
- ▶ **Secure Data**  
This is the data that needs to be encrypted before it is stored.

Upon invocation, the command calls the processing program (SET\_DATA), which controls the command function. The functions performed include testing the required key, encrypting the data, and storing it.

## Understanding the SET\_DATA command

**Note:** This section provides a detailed and structural analysis of the SET\_DATA command. If you want to quickly run the entire application to have a complete view, you may skip to 11.3.4, “Decrypt data on source system: GET\_DATA command” on page 185. However, we strongly recommend that you to come back to this point and read the rest of section carefully to understand the logic behind this.

Our SET\_DATA command requires that the user provide information about the keystores, the key that we wish to use, and, of course, the customer data. We supply the information shown in Figure 11-17.

```
Set Customer Data (SET_DATA)

Type choices, press Enter.

Key LABEL of Key record (KL) . . > LABEL_04
Key Store File Souce (KSFS) . . KEYFILES      Name
  Library Name . . . . . R7399_API      Name, *CURLIB
Key Store File Target (KSFT) . . KEYFILET   Name
  Library Name . . . . . R7399_API      Name, *CURLIB
Data File (DF) . . . . . ENCDATA        Name
  Library Name . . . . . R7399_API      Name, *CURLIB
First Name . . . . . > Tom
Last Name . . . . . > Bliss
Address . . . . . > '1232 Maple St.'
City . . . . . > Anytown
Country . . . . . > USA
State . . . . . > MN                    Character value
ZIP Code . . . . . > 25611              Character value
Phone Number . . . . . > 32565          Character value
Secure Data . . . . . > 'This is secret'
```

Figure 11-17 The Set Customer Data (SET\_DATA) CL command prompt

The command passes the parameter data to the command processing program SET\_DATA. This program calls the ENC\_DATA program to encrypt and store the data in the ENCDATA file. At the end, the program displays a completion message, as shown in Figure 11-18.

```
Selection or command
====>

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu
Key Label LABEL_04 in R7399_API/KEYFILES was used to encrypt data
```

Figure 11-18 Completion message about data encryption function

The overall structure of the SET\_DATA command function is depicted in Figure 11-19.

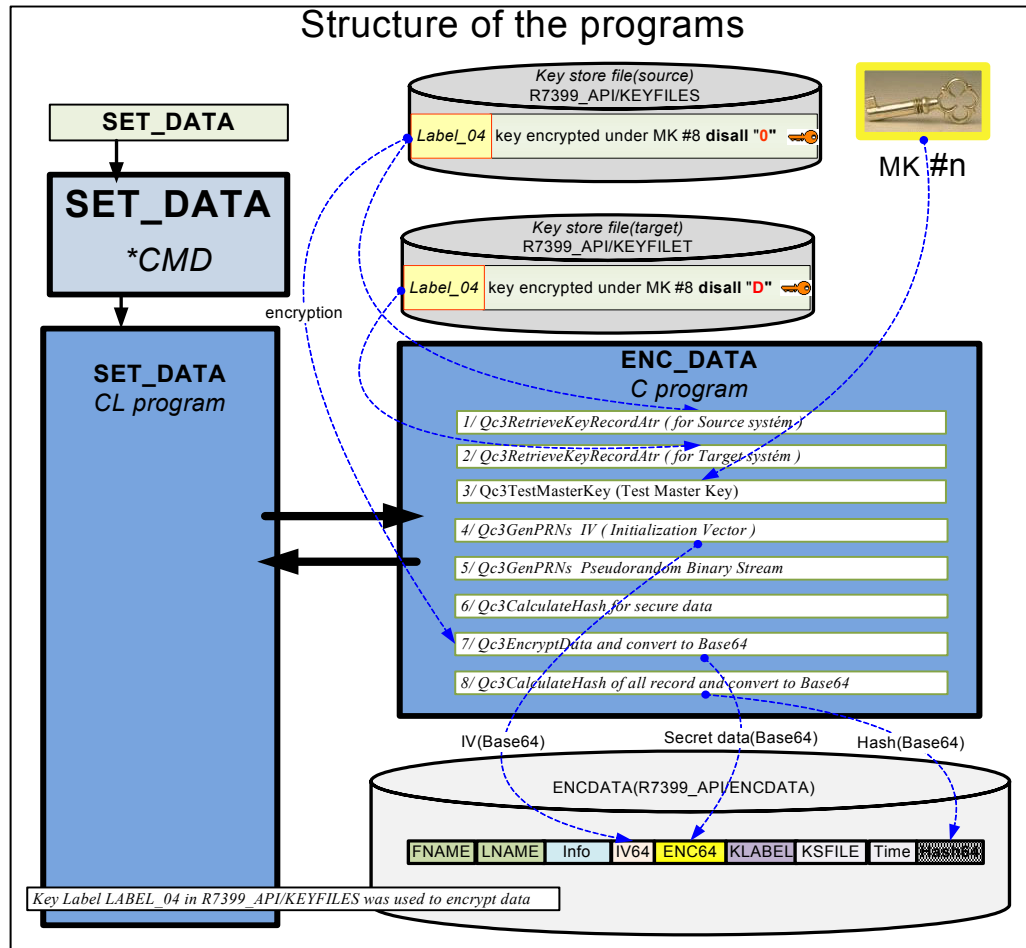


Figure 11-19 Structure of Set Customer Data (SET\_DATA) CL command

## Used APIs

To encrypt and store data, we utilize the APIs discussed in this section.

### Retrieve Key Record Attributes API

This API is *Qc3RetrieveKeyRecordAtr* in ILE and *QC3RTVKA* in OPM.

The first step in our application's encryption process is to check the existence of the key with the label "LABEL\_04" in the keystore file KEYFILES and KEYFILET. We retrieve the attributes of the key in the keystore file (key type, key size, Master key ID, Master key verification value, and Disallowed function). The retrieval is performed by the Retrieve Key Record Attributes API, as depicted in Example 11-16 for the source system and Example 11-17 on page 174 for the target system.

Example 11-16 *Qc3RetrieveKeyRecordAtr* API for the Source system

```
Qc3RetrieveKeyRecordAtr( kscopy.Key_Store,
                        endout.KEYLAB,
                        &keyType_rtv,
                        &keySize_rtv,
                        &mkid_rtv,
                        &Master_verification_value_rtv,
```

```

        &disFunc_rtv,
        &errCode);

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3RetrieveKeyRecordAtr \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    memcpy ( argv[5], "1", 1 );
    return;
}

```

---

*Example 11-17 Qc3RetrieveKeyRecordAtr API for the target system*

---

```

Qc3RetrieveKeyRecordAtr( T_kskey.Key_Store,
                        endaout.KEYLAB,
                        &T_keyType_rtv,
                        &T_keySize_rtv,
                        &T_mkid_rtv,
                        &T_Master_verification_value_rtv,
                        &T_disFunc_rtv,
                        &errCode);

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3RetrieveKeyRecordAtr \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    memcpy ( argv[5], "1", 1 );
    return;
}

```

---

The parameter list includes:

- ▶ kskey.Key\_Store, T\_kskey.Key\_Store  
This is the qualified name of the keystore file containing the key.
- ▶ endaout.KEYLAB  
This is the key label.
- ▶ keyType\_rtv, T\_keyType\_rtv  
This is the algorithm type of key and is returned by the API.
- ▶ keySize\_rtv, T\_keySize\_rtv  
This is the size of the key and is returned by the API.
- ▶ mkid\_rtv, T\_mkid\_rtv  
This is the master key ID used to secure the keystore file and is returned by the API.
- ▶ Master\_verification\_value\_rtv, T\_Master\_verification\_value\_rtv  
This is the KVV for the master key at the time that the key was encrypted. This can be compared to the current master key KVV to determine whether the key needs to be translated.
- ▶ disFunc\_rtv, T\_disFunc\_rtv  
This is the disallowed function setting for the key and is returned by the API.

► `errCode`

Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp?topic=/apis/qc3rtvka.htm>

For our scenario we test that they key type, the key size, the Master key ID, and the Master Key Verification Value are same in the keystore file as in the source and target systems. The value of the disallowed function for the key on the source system should be all function allowed "0", and for the target system should be only decryption is allowed "13"  
Example 11-18.

*Example 11-18 Comparison of the key record attributes*

---

```
if ( keyType_rtv    != T_keyType_rtv
    ||
    keySize_rtv    != T_keySize_rtv
    ||
    mkid_rtv       != T_mkid_rtv
    ||
    disFunc_rtv    != Qc3_Disall_none
    ||
    T_disFunc_rtv  != Qc3_All_dec_D
    ||
    memcmp ( Master_verification_value_rtv,
             T_Master_verification_value_rtv,
             sizeof(Master_verification_value_rtv)) != 0 )
{
    printf (">>Qc3RetrieveKeyRecordAtr - compare \n");
    memcpy ( argv[5], "1", 1 );
    return;
}
```

---

### **Test Master Key API**

This API is *Qc3TestMasterKey* in ILE and *QC3TSTMK* in OPM.

The master key verification value for the key on the source and target systems should be the same value as the key verification value retrieved from the corresponding master key (Example 11-19).

The retrieval is performed by the Test Master Key API, as depicted in Example 11-19.

*Example 11-19 Retrieval of the key verification value for the specified master key*

---

```
mstkv[0] = Qc3_MK_Current;

Qc3TestMasterKey (&mkid_rtv,
                 &mstkv,
                 &Master_verification_value_rtv,
```

```

        &errCode);

if ( errCode.Bytes_Available != 0 )
{
    printf (">>QcTestMasterKey\n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    memcpy ( argv[5], "1", 1 );
    return;
}

```

---

*Example 11-20 Comparison of the master key verification value*

---

```

if ( memcmp ( Master_verification_value_rtv,
             T_Master_verification_value_rtv,
             sizeof(Master_verification_value_rtv)) != 0 )
{
    printf (">>QcTestMasterKey - compare\n");
    memcpy ( argv[5], "1", 1 );
    return;
}

```

---

### **Generate Pseudorandom Number String API**

This API is *Qc3GenPRNs* in ILE and *QC3GENRN* in OPM.

This API generates a pseudorandom binary stream, which we use for an Initialization Vector (IV). The call to the Generate Pseudorandom Numbers API is depicted in Example 11-21.

*Example 11-21 Qc3GenPRNs API*

---

```

PRNtype = Qc3PRN_TYPE_NORMAL;      /* Generate real random numbers*/
PRNparity = Qc3PRN_NO_PARITY;      /* Do not adjust parity      */
PRNlen = 16;                       /* Generate 16 bytes        */

```

```

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

```

```

Qc3GenPRNs( IV,
           PRNlen,
           PRNtype,
           PRNparity,
           &errCode);

```

```

if ( errCode.Bytes_Available != 0 )
{ printf (">>Qc3GenPRNs \n");
  printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
  memcpy ( argv[4], "1", 1 );
  return;
}

```

---

The parameter list includes:

► IV

This is the 16-byte binary stream generated by the API.

► PRNlen

This is the number of bytes that we wish to have the API return. We are generating a 128-bit stream (16 x 8 = 128 bit).

► PRNtype

The indicates whether the API is to generate a real or a test binary stream. We want a real stream (the test stream uses preset values for key and seed).

► PRNParity

This specifies whether the API should modify the low-order bit to a specific parity. We request no adjustment to the parity bit.

► errCode

Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

### **Generate Pseudorandom Number String API**

**Note:** This API is used to generate a random string to be used with secret data to provide stronger security.

The second call to the Generate Pseudorandom Numbers API is depicted in Example 11-22.

*Example 11-22 Generate binary stream*

---

```
memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
PRNlen = sizeof(Security_Data) - sizeof(HASH);

Qc3GenPRNs( Security_Data,
            PRNlen,
            PRNtype,
            PRNparity,
            &errCode);

if (errCode.Bytes_Available != 0) {
    printf(">>Qc3GenPRNs \n");
    printf("  errCode.Exception_Id = %.7s\n",errCode.Exception_Id);
    memcpy ( argv[4], "1", 1 );
    return;
}

memcpy (&Security_Data[30], Security_Data, PRNlen);
memcpy (&Security_Data[45], Security_Data, PRNlen);

for(i=0;i<14;i++) {
    Security_Data[38+i] = Security_Data[16+i]^Security_Data[i];
}

```

---

The secure field now looks similar to the data in Figure 11-20.

EVAL Security_Data: x 80					
00000	18D4E507	445185BF	AB9964EA	93964915	- .MV.ãëë×ìrÃ?loñ.
00010	E38889A2	4089A240	A2858399	85A318D4	- This is secret.M
00020	E5074451	85BFFB5C	6CA504D8	27FF091C	- V.ãëë×Û*%v.Q...
00030	E7731635	BFAB9964	EA939649	15000000	- XË..×ìrÃ?loñ....
00040	00000000	00000000	00000000	00000000	- .....

Figure 11-20 Sample data including pseudorandom binary stream

### Calculate Hash API

This API is *Qc3CalculateHash* in ILE and *QC3CALHA* in OPM.

This API is used to generate a secure hash of the secure data (including the pseudorandom data binary stream). Calling the Calculate Hash API is depicted in Example 11-23.

Example 11-23 Calculate hash for secure data and hints

```

csp = Qc3_Any_CSP; /* Use any crypto provider
Length_of_input_data = sizeof(Security_Data) - sizeof(HASH);
memset ( Algorithm_description, 0, sizeof(Algorithm_description));
Algorithm_description[3] = Qc3_SHA1;

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
Qc3CalculateHash ((char*)&Security_Data,
                  &Length_of_input_data,
                  "DATA0100",
                  (char*)&Algorithm_description,
                  "ALGD0500",
                  &csp,
                  NULL,
                  Security_Data + Length_of_input_data,
                  &errCode );

if (errCode.Bytes_Available != 0) {
    printf(">>Qc3CalculateHash \n");
    printf (" errCode.Exception_Id = %.7s\n",errCode.Exception_Id);
    memcpy ( argv[4], "1", 1 );
    return;
}

```

The parameter list includes:

- ▶ Security\_Data
  - This is the data provided by the user on the command, combined with the pseudorandom binary stream.
- ▶ Length\_of\_input\_data
  - This is the length of the security data.
- ▶ "DATA0100"
  - This indicates to the API which format we wish to use. This format indicates that the Security\_Data parameter contains the data to be hashed rather than an array of pointers.



- ▶ **Algorithm\_description**  
For SHA-1, this will be binary 00000002.
- ▶ **“ALGD0500”**  
This indicates to the API which format we wish to use for the Algorithm\_description parameter. For SHA-1, this will be binary 00000002.
- ▶ **csp**  
This indicates which cryptographic service provider will perform the hash operation—software or hardware. We instruct i5/OS to select the appropriate CSP.
- ▶ **Security\_Data + Length\_of\_input\_data**  
This is the area in which to store the returned hash value. The length of the hash is defined by the hash algorithm. For SHA-1 this will be a character value of 20 bytes.
- ▶ **errCode**  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:  
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp>  
Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

After the hashing function, the Security\_Data field appears in Figure 11-21.

EVAL Security_Data: x 80	
00000	18D4E507 445185BF AB9964EA 93964915 - .MV.ãëex¿rÃ?loñ.
00010	E38889A2 4089A240 A2858399 85A318D4 - This is secret.M
00020	E5074451 85BFFB5C 6CA504D8 27FF091C - V.ãëexÛ*%v.Q....
00030	E7731635 BFAB9964 EA939649 15669545 - XË..x¿rÃ?loñÃÃnã
00040	<b>B5C85997 3A713C78 0A7A0C12 5C73D507</b> - §HBp.É.Ï.:.*.ËN.

Figure 11-21 Security\_Data field after hashing

### Encrypt Data API

This API is *Qc3EncryptData* in ILE and *QC3ENCDDT* in OPM.

This API scrambles data into an unintelligible form. In this case, the key is stored in a keystore file. The call to the Encrypt Data API is depicted in Example 11-24.

Example 11-24 Qc3EncryptData API

```
memset(&algD, 0, sizeof(algD)); /* Init alg description to null*/
algD.Block_Cipher_Algo = Qc3_AES; /* Set AES algorithm */
algD.Block_Length = 16; /* Block size is 16 */
algD.Mode = Qc3_CBC; /* Use cipher block chaining */
algD.Pad_Option = Qc3_No_Pad; /* Do not pad */
memcpy(algD.Init_Vector, IV, 16); /* Copy IV to alg description */

plainLen = sizeof(Security_Data);
cipherLen = sizeof(Encrypted_Data);

csp = Qc3_Any_CSP; /* Use any crypto provider */

memset(&errCode, 0, sizeof(errCode));
```

```

errCode.Bytes_Provided = sizeof(errCode);
Qc3EncryptData( Security_Data,
                &plainLen,
                Qc3_Data,
                (char*)&algD,
                Qc3_Algorithm_Block_Cipher,
                &kskey,
                Qc3_Key_KSLabel,
                &csp,
                NULL,
                Encrypted_Data,
                &cipherLen,
                &rtLen,
                &errCode);

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3EncryptData \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    memcpy ( argv[4], "1", 1 );
    return;
}

```

---

The parameter list includes:

- ▶ **Security\_Data**  
This is the data to be encrypted.
- ▶ **plainLen**  
This is either the length of the data to be encrypted or the number of elements to be encrypted, depending on the format supplied in the **Qc3\_Data** parameter.
- ▶ **Qc3\_Data**  
This indicates that the **Security\_Data** parameter contains the actual data to be encrypted rather than an array of pointers to the data to be encrypted.
- ▶ **algD**  
This is the algorithm and associated parameters for encrypting the data. The format of this parameter is specified in the **Qc3\_Algorithm\_Block\_Cipher** parameter.
- ▶ **Qc3\_Algorithm\_Block\_Cipher**  
This is the format of the algorithm description. In this case, it indicates that the hash algorithm is specified.
- ▶ **kskey**  
This is the key and associated parameters for encrypting the data. The format of this parameter is specified in the **Qc3\_Key\_KSLabel** parameter.
- ▶ **Qc3\_Key\_KSLabel**  
This is the format of the key description. In this case, it specifies that a keystore file and label are supplied.
- ▶ **csp**  
This indicates which cryptographic service provider will perform the encryption operation—software or hardware. Using a character value of 0 instructs i5/OS to select the appropriate CSP.

- ▶ NULL (cryptographic device name)  
This is the associated hardware device if a hardware CSP is selected. Otherwise, this parameter must be blank or null.
- ▶ Encrypted\_Data  
This is the encrypted data returned by the API.
- ▶ cipherLen  
This is the length of the Encrypted\_Data parameter.
- ▶ rtnLen  
This is the length of the encrypted data returned by the API in the Encrypted Data parameter.
- ▶ errCode  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:  
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp>  
Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

After the encryption function, the Encrypted\_Data field appears as shown in Figure 11-22.

EVAL Encrypted_Data: x 80	
00000	28021087 3DE95597 E8DC0691 C8EB3FDC - ...g.ZípYü.jHÔ.ü
00010	10A1BC33 6CE40EE3 FEBB1FC1 71D7D903 - .~%.%U.TÚ].AÉPR.
00020	91496EB6 929E3839 B00087F8 D9327F82 - jñ>¶kÆ..^.g8R."b
00030	B4BE8852 F62074F0 0741C0F2 87FD35EF - ©`hè6.Ë0. {2gÛ.Û
00040	82497726 A55A8EB2 4B8E1911 D60DC87B - bñÏ.v!p¥.p..0.H#

Figure 11-22 Encrypted\_Data field after encryption

### Convert encrypted field and IV record to Base64 form

This step is to convert the encrypted field and IV record to Base64 form so that it is readable (Example 11-25).

Example 11-25 Encrypted data and IV to Base64

---

```
len_Base64 = to_Base64 ( IV, sizeof(IV),endaout.IV64, len_Base64 );
len_Base64 = to_Base64( Encrypted_Data, rtnLen, endaout.ENC64, len_Base64 );
```

---

### Calculate HASH API

This API is *Qc3CalculateHash* in ILE and *QC3CALHA* in OPM.

This API is used to generate a secure hash of the entire record (including the plain and encrypted data). The call to the Calculate Hash API is as depicted in Example 11-26.

Example 11-26 Qc3CalculateHash API for all record

---

```
memset ( Algorithm_description, 0, sizeof(Algorithm_description));
Algorithm_description[3] = Qc3_SHA1;

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
```

```

Qc3CalculateHash ((char*)&endaout.FNAME,
                  &Length_of_input_data,
                  "DATA0100",
                  (char*)&Algorithm_description,
                  "ALGD0500",
                  &csp,
                  NULL,
                  HASH,
                  &errCode );

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3CalculateHash \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    memcpy ( argv[4], "1", 1 );
    return;
}

```

---

The parameter list includes:

- ▶ `endaout.FNAME`  
This is the encrypted customer data record.
- ▶ `Length_of_input_data`  
This is the length of the encrypted customer data record.
- ▶ "DATA0100"  
This indicates to the API the format of the input data. This format indicates that the `endaout.FNAME` parameter contains the data to be hashed, rather than an array of pointers.
- ▶ `Algorithm_description`  
For SHA-1, this will be binary 00000002.
- ▶ "ALGD0500"  
This indicates to the API the format of the `Algorithm_description` parameter. For SHA-1, this will be binary 00000002.
- ▶ `csp`  
This indicates which cryptographic service provider will perform the hash operation—software or hardware. We instruct i5/OS to select the appropriate CSP.
- ▶ `HASH`  
This is the area in which to store the returned hash value. The length of the hash is defined by the hash algorithm. For SHA-1 this will be a character value of 20 bytes.
- ▶ `errCode`  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

## Storing customer data in file ENCDATA

The final step is to store the record in the customer file (ENCDATA). This is depicted in Example 11-27.

*Example 11-27 Store customer data to ENCDATA file*

---

```

if (( endaPtr = _Ropen( DataFile, "rr+", riofb=N")) == NULL) {
    printf (">>_Ropen ENCDATA \n");
    printf ("  Open of ENCDATA file failed errno %d.\n",errno );
    memcpy ( argv[5], "1", 1 );
    _Rclose(endaPtr);
    return;
}
if ((_Rwrite( endaPtr , &endaout,
              sizeof(endaout))->num_bytes < sizeof(endaout)) {
    printf (">>_Rwrite ENCDATA \n");
    printf ("  _Rwrite of ENCDATA file failed errno %d.\n",errno );
    memcpy ( argv[5], "1", 1 );
    _Rclose(endaPtr);
    return;
}
_Rclose(endaPtr);
return;

```

---

### What the customer record looks like

We assume that all of the information needed to decrypt data will be stored in a record in the ENCDATA file (Figure 11-23), and that the file will be secured correctly using i5/OS object-level security. In our example, the keystore files for the source and target systems, as well as the ENCDATA data file, were created with public authority \*EXCLUDE. Access to this file may be accomplished via private or adopted authority to both files.

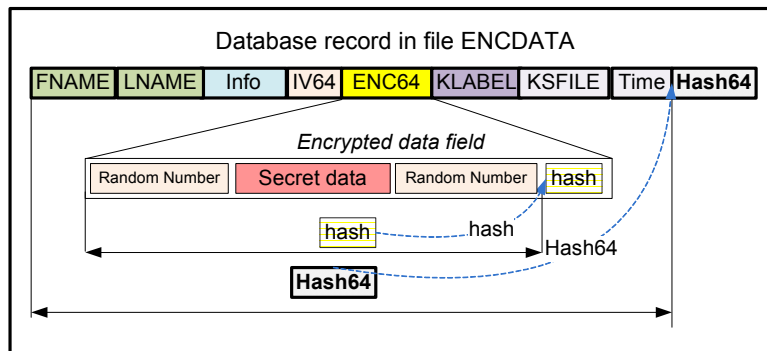


Figure 11-23 Record structure

The record includes Example 11-28 on page 184:

- ▶ First name (FNAME)
  - This is also an index field.
- ▶ Last name (LNAME)
  - This is also an index field.
- ▶ Information data (ADRES, CITY, COUNTRY, STATE, ZIP, PHONE)

- ▶ Initialization Vector (IV)  
This value is converted to Base64 format to permit display.
- ▶ Private customer data,  
This data is encrypted and converted to Base64 format to permit display. The string comprises:
  - Pseudorandom number binary stream.
  - Secure data
  - Hash of pseudorandom number, secure data
- ▶ Key Label LABEL\_04
- ▶ Keystore file name for target system
- ▶ The time stamp of the record
- ▶ HASH of the entire record  
The hash is also converted to Base64 format to permit display.

The customer record data is depicted in Example 11-28.

*Example 11-28 Record in database file ENCDATA*

---

```

EVAL endout: x 340
00000 E3969440 40404040 40404040 404040C2 - Tom B
00010 9389A2A2 40404040 40404040 4040F1F2 - liss 12
00020 F3F240D4 81979385 40E2A34B 40404040 - 32 Maple St.
00030 4040C195 A8A396A6 95404040 40404040 - Anytown
00040 40404040 4040E4E2 C1404040 40404040 - USA
00050 40404040 40404040 4040D4D5 F2F5F6F1 - MN2561
00060 F1F3F2F5 F6F591C3 A797C3F8 F2C48888 - 132565jCxpC82Dhh
00070 C2D783F9 96F281E7 F1C8D1A6 7E7ED2C1 - BPc9o2aX1HJw==KA
00080 C9D888A9 F397E5E9 8696F3C1 81D9A8D6 - IQhz3pVZfo3AaRy0
00090 A261F3C2 C388A5C4 D5A2F5C1 F7916199 - s/3BChvDNs5A7j/r
000A0 A286A6E7 C8E7F2D8 D6D9E2E6 F6F29297 - sfwXHX2QORSW62kp
000B0 F4F4D682 C1C18861 91E9D495 4EC3A3D3 - 440bAAh/jZMn+CtL
000C0 F6C9E4A5 E88784D7 C1C8D883 C4A88861 - 6IUvYgdPAHQcDyh/
000D0 F0F1F7F4 D1D184A8 8193E696 F6A8E2F4 - 0174JJdya1Wo6yS4
000E0 F4E9C584 E8D5A8C8 A27ED3C1 C2C5D36D - 4ZEdYNyHs=LABEL_
000F0 F0F44040 40404040 40404040 40404040 - 04
00100 40404040 40404040 4040D2C5 E8C6C9D3 - KEYFIL
00110 C5E34040 D9F7F3F9 F96DC16D D7C94040 - ET R7399_API 20
00120 F0F760F0 F560F2F0 60F1F94B F1F24BF0 - 07-05-20-19.12.0
00130 F04BF8F8 F0F0F0F0 A286E993 D1A6F982 - 0.880000sfZ1Jw9b
00140 A6A7E296 A2C1E6A7 97979998 88E382F7 - wxSosAWxpprqhTb7
00150 F5A3877E ..... - 5tg=.....

```

---

## Running SET\_DATA command step summary

We have now taken customer data (some public, some private) and encrypted and stored it in a database file (ENCDATA). The process that we used is depicted in Figure 11-24.

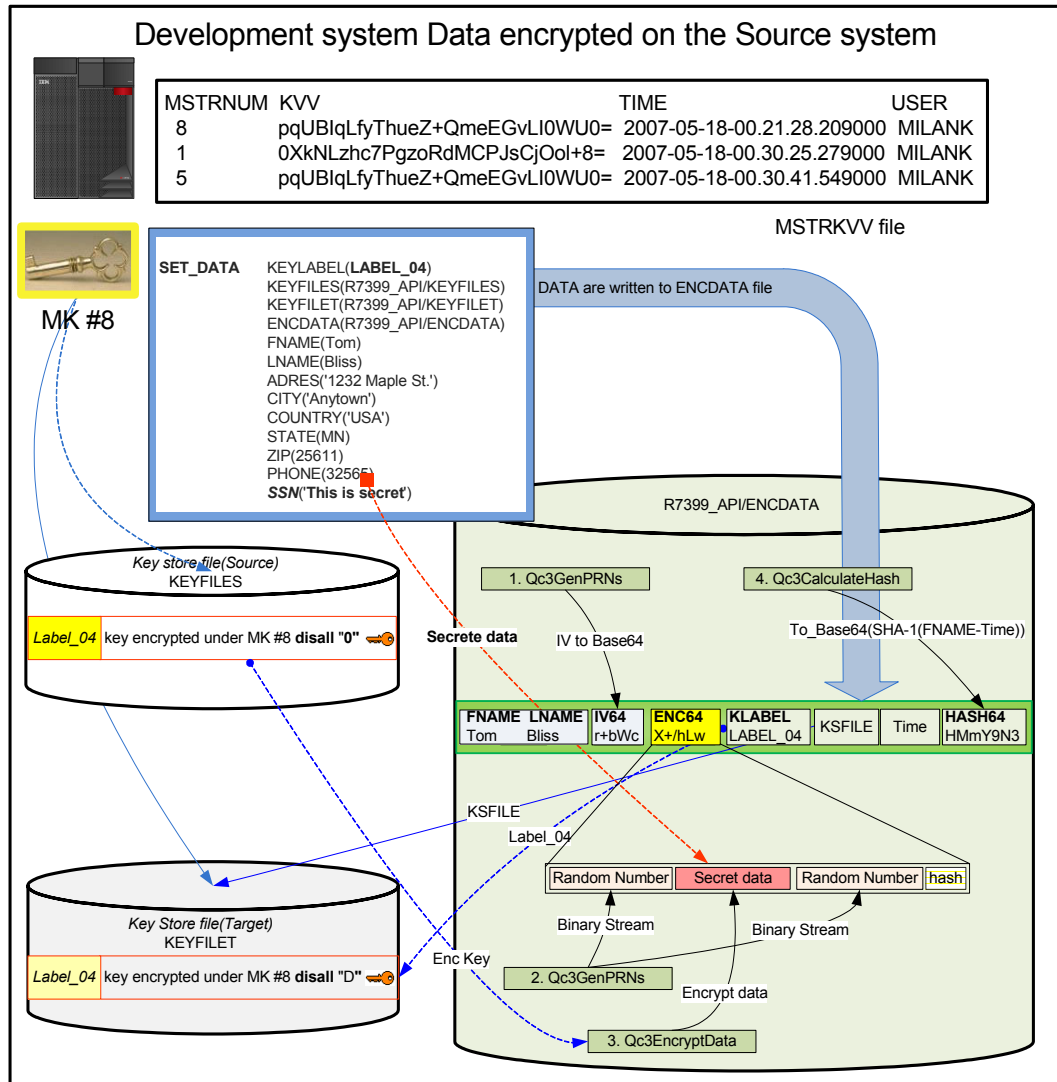


Figure 11-24 Encryption scenario

### 11.3.4 Decrypt data on source system: GET\_DATA command

This section provides the steps to encrypt/decrypt the secret data in the ENCDATA database file on the same system. We have now encrypted data in the database file ENCDATA and we want to test whether the information was stored in good order.

## Executing GET\_DATA command

This command reads data (including private information) from the database file ENCDATA on the source system, and appears as in Figure 11-25. Although this type of function is more commonly performed via an application program with a display format, we created a command-based interface to allow us to focus on the cryptographic functionality.

```
Get Data (GET_DATA)

Type choices, press Enter.

Data File (DF) . . . . . ENCDATA      Name
Library Name . . . . . R7399_API     Name, *CURLIB
First Name (FN) . . . . . Tom
Last Name (LN) . . . . . Bliss
```

Figure 11-25 Command Get Data

The command requires you to supply the following information:

- ▶ Data file and library name

We specify the fully qualified file name of the database file where the data was stored.

- ▶ First and last name

The program uses the first and the last name as the database key field of the database file and reads from this database file record. The structure of this record is shown in Figure 11-23 on page 183.

Upon invocation, the command calls the processing program (GET\_DATA) that controls the command function. The overview of this function is to check object existence of the database file, and then call the program (DEC\_DATA) to obtain the secret information of the desired customer.

## Understanding GET\_DATA command

**Note:** This section provides a detailed and structural analysis of the GET\_DATA command. If you want to quickly run the entire application to have a complete view, you may skip to 11.3.5, “Decrypt data on target system” on page 194. However, we strongly recommend the you to come back to this point and read the rest of section carefully to understand the logic behind this.

This section provides background information about the GET\_DATA command.

### ***Decrypting data to obtain desired information***

Our GET\_DATA command requires that the user provide the fully qualified file name of the database file where the desired information was stored, and, of course, the first and the last name of the customer.



We supply the information as shown in Figure 11-26.

```
Get Data (GET_DATA)

Type choices, press Enter.

Data File (DF) . . . . . ENCDATA      Name
Library Name . . . . . R7399_API     Name, *CURLIB
First Name . . . . . Tom
Last Name . . . . . Bliss
```

Figure 11-26 Command Get Data

The command passes the parameter data to the command processing program GET\_DATA. This program calls the GET\_DATA program to check for the existence of the file ENCDATA and then calls the program DEC\_DATA to obtain secret information of the desired customer. At the end, the program displays a completion message, as shown in Figure 11-27.

```
SecureData = This is secret
Press ENTER to end terminal session.
===>

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=
F23=Set initial menu
Data was decrypted
```

Figure 11-27 Message about decryption data

The overall structure of the GET\_DATA command function is depicted in Figure 11-28.

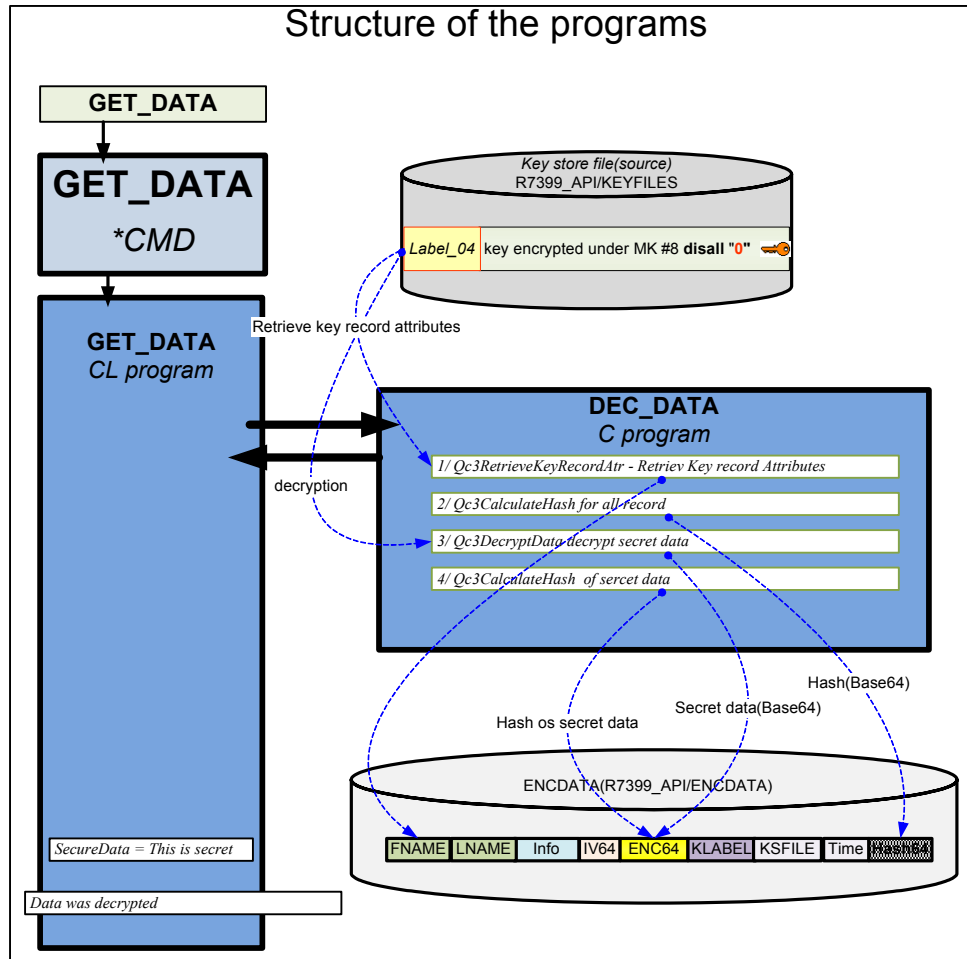


Figure 11-28 Structure of Get Customer Data (GET\_DATA) CL command

### Retrieving record from ENCDATA file

The program DEC\_DATA uses the first name and the last name as the database key field to retrieve the record from the ENCDATA file (Example 11-29).

#### Example 11-29 Retrieve record from ENCFIL

```
memcpy( Data_File, argv[1], sizeof(Data_File));
memset( DataFile, 0, sizeof(DataFile));

for ( i=10,j= 0; i < 20 && Data_File[i] != ' ';
      DataFile[j] = Data_File[i], ++i, ++j);
DataFile[j] = '/';
++j;
for ( i= 0 ; i < 10 && Data_File[i] != ' ';
      DataFile[j] = Data_File[i], ++i, ++j);

memcpy( endakey.FNAME, argv[2], sizeof(endaout.FNAME));
memcpy( endakey.LNAME, argv[3], sizeof(endaout.LNAME));
memcpy( argv[3], " ", 1 );
if (( endaPtr = _Ropen( DataFile, "rr+", riofb=N")) == NULL) {
    printf (">>_Ropen ENCDATA \n");
}
```

```

        printf ("  Open of ENCDATA file failed errno %d.\n" , errno );
        memcpy ( argv[4], "1", 1 );
        _Rclose(endaPtr);
        return;
    }
    dbendaPtr = _Rreadk( endaPtr, &endain, sizeof(endain), __KEY_EQ,
        &endakey, sizeof(endakey));
    if ( endaPtr->riofb.num_bytes == 0 ) {
        printf (">>Rreadk ENCDATA \n");
        printf ("  No data %.15s in ENCDATA\n", endaout.FNAME);
        memcpy ( argv[4], "1", 1 );
        _Rclose(endaPtr);
        return;
    }
    _Rclose(endaPtr);

```

---

## Used APIs

To decrypt data, we utilize the APIs discussed in this section.

### ***Retrieve Key Record Attributes API***

This API is *Qc3RetrieveKeyRecordAtr* in ILE and *QC3RTVKA* in OPM.

The Retrieve Key Record Attributes API retrieves the key type and key size of a key that is stored in a keystore file. Using key label KLABEL and the name of the keystore file KSFIL written into the record of database file ENCDATA, we retrieve key record attributes. The retrieval is performed by the Retrieve Key Record Attributes API, as depicted in Example 11-30. This is for testing purposes for our system to ensure that the Key Label entry exists in the keystore file.

#### *Example 11-30 Qc3RetrieveKeyRecordAtr API*

---

```

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

Qc3RetrieveKeyRecordAtr( endain.QKSFN,
                        endain.KEYLAB,
                        &keyType_rtv,
                        &keySize_rtv,
                        &mkid_rtv,
                        Master_verification_value_rtv,
                        &disFunc_rtv,
                        &errCode);

if ( errCode.Bytes_Available != 0 ) {
    printf (">>Qc3RetrieveKeyRecordAtr \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return;
}

```

---

For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iseri/v5r4/index.jsp?topic=/apis/qc3rtvka.htm>

### **Calculate Hash API**

This API is *Qc3CalculateHash* in ILE and *QC3CALHA* in OPM.

This API is used to generate a secure hash of the entire record (including the plain and encrypted data). The call to the Calculate Hash API is depicted in Example 11-31.

#### *Example 11-31 Calculate Hash for entire record*

---

```
csp = Qc3_Any_CSP;
Length_of_input_data = sizeof(endaout) - sizeof(endaout.HASH64);

memset ( Algorithm_description, 0, sizeof(Algorithm_description));
Algorithm_description[3] = Qc3_SHA1;
memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

Qc3CalculateHash ((char*)&endain.FNAME,
                  &Length_of_input_data,
                  "DATA0100",
                  (char*)&Algorithm_description,
                  "ALGD0500",
                  &csp,
                  NULL,
                  HASH,
                  &errCode );

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3CalculateHash \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    memcpy ( argv[4], "1", 1 );
    return;
}
```

---

We transform this hash from a binary stream to Base64 format and compare the value with the written value in the database file ENCDATA, as shown in Example 11-32.

#### *Example 11-32 Compare hash value of the entire record*

---

```
len_Base64 = to_Base64( HASH, sizeof(HASH), endaout.HASH64, len_Base64 );
if ( memcmp( endaout.HASH64, endain.HASH64, len_Base64) != 0 )
{
    printf (">>HASH record error \n");
    memcpy ( argv[4], "1", 1 );
    return ERROR;
}
```

---

### **Decrypt Data API**

This API is *Qc3DecryptData* in ILE and *QC3DECDDT* in OPM.

This API restores encrypted data to a clear (intelligible) form. If both hash values are the same, the program converts IV from Base64 to binary and decrypts data with the Decrypt Data API, depicted in Example 11-33.

*Example 11-33 Qc3DecryptData API for DEC\_DATA*

---

```

memset(&algD, 0, sizeof(algD));      /* Init alg description to null*/
algD.Block_Cipher_Algo = Qc3_AES;   /* Set AES algorithm          */
algD.Block_Length = 16;             /* Block size is 16          */
algD.Mode = Qc3_CBC;                /* Use cipher block chaining */
algD.Pad_Option = Qc3_No_Pad;       /* Do not pad                 */

len_text = Base64_to( endain.IV64 , sizeof(endain.IV64),
                    algD.Init_Vector, len_text);
len_text = Base64_to( endain.ENC64, sizeof(endain.ENC64), n_Encrypted len_text);
cipherLen = sizeof(n_Security);

memset(&kskey, '\0', sizeof(kskey));
memcpy( kskey.Key_Store, endain.QKSFN, sizeof(endain.QKSFN));
memcpy( kskey.Record_Label, endain.KEYLAB, sizeof(endain.KEYLAB));

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
Qc3DecryptData( n_Encrypted,
                &len_text,
                (char*)&algD,
                Qc3_Algo_Block_Cipher,
                &kskey,
                Qc3_Key_KSLabel,
                &csp,
                NULL,
                n_Security,
                &cipherLen,
                &rtLen,
                &errCode);

if ( errCode.Bytes_Available != 0 ) {
    printf (">>Qc3EncryptData \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    memcpy ( argv[4], "1", 1 );
    return;
}

```

---

The parameter list includes:

- ▶ **n\_Encrypted**  
This is the data to be decrypted.
- ▶ **len\_text**  
This is the length of the encrypted data.
- ▶ **algD**  
The algorithm and associated parameters for decrypting the data. The format of the algorithm description is specified by the Qc3\_Algo\_Block\_Cipher format name parameter.

- ▶ **Qc3\_Alg\_Block\_Cipher**  
The algorithm description format name indicates the format of the algorithm description.
- ▶ **kskey**  
This is the key and associated parameters for decrypting the data. The format of this parameter is specified in the **Qc3\_Key\_KSLabel** parameter.
- ▶ **Qc3\_Key\_KSLabel**  
This is the key description format name. In this case, it specifies that a keystore file and label are supplied.
- ▶ **csp**  
This indicates which cryptographic service provider will perform the encryption operation—software or hardware. Using a character value of 0 instructs i5/OS to select the appropriate CSP.
- ▶ **NULL**  
This is the associated hardware device if a hardware CSP is selected. Otherwise, this parameter must be blank or null.
- ▶ **n\_Security**  
This is the clear data returned by the API.
- ▶ **cipherLen**  
This is the length of area provided for the clear data. To ensure sufficient space, specify an area at least as large as the length of encrypted data. If the length of area provided for the clear data is too small, an error will be generated and no data will be returned in the clear data parameter.
- ▶ **rtLen**  
Length of clear data returned.
- ▶ **errCode**  
Error code I/O Char(\*). The structure in which to return error information.  
  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:  
  
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp>  
  
Then select **Programming** → **Application programming interfaces** → **API concepts**.

### **Calculate Hash API**

This API is used again to generate a secure hash of the encrypted part of the record. The call to the Calculate Hash API is depicted in Example 11-34.

*Example 11-34 Calculate hash for encrypted data*

---

```

Length_of_input_data = sizeof(n_Security) - sizeof(HASH);
memset ( Algorithm_description, 0, sizeof(Algorithm_description));
Algorithm_description[3] = Qc3_SHA1;

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);
Qc3CalculateHash ((char*)&n_Security,
                  &Length_of_input_data,
                  "DATA0100",

```

```

        (char*)&Algorithm_description,
        "ALGD0500",
        &csp,
        NULL,
        HASH,
        &errCode );

if ( errCode.Bytes_Available != 0 ) {
    printf (">>Qc3CalculateHash \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    memcpy ( argv[4], "1", 1 );
    return;
}

```

---

### **Verification process**

An equal comparison of calculated hash and the hash retrieved from the record confirms that the secret data that they possess is the same as that which we encrypted see Example 11-35.

#### *Example 11-35 Comparison of the hash*

---

```

if ( memcmp(&n_Security[60], HASH, sizeof(HASH)) != 0 ) {
    printf (">>HASH error secret data \n");
    memcpy ( argv[4], "1", 1 );
    return;
}

```

---

If both values are the same, the secret data is displayed as in Figure 11-29.

<pre> SecureData = <b>This is secret</b> Press ENTER to end terminal session. </pre>
--

*Figure 11-29 Secret data*

### **Running GET\_DATA command step summary**

We have now taken secret customer data from the database file (ENCDATA) and decrypted it.

The process that we used is depicted in Figure 11-30.

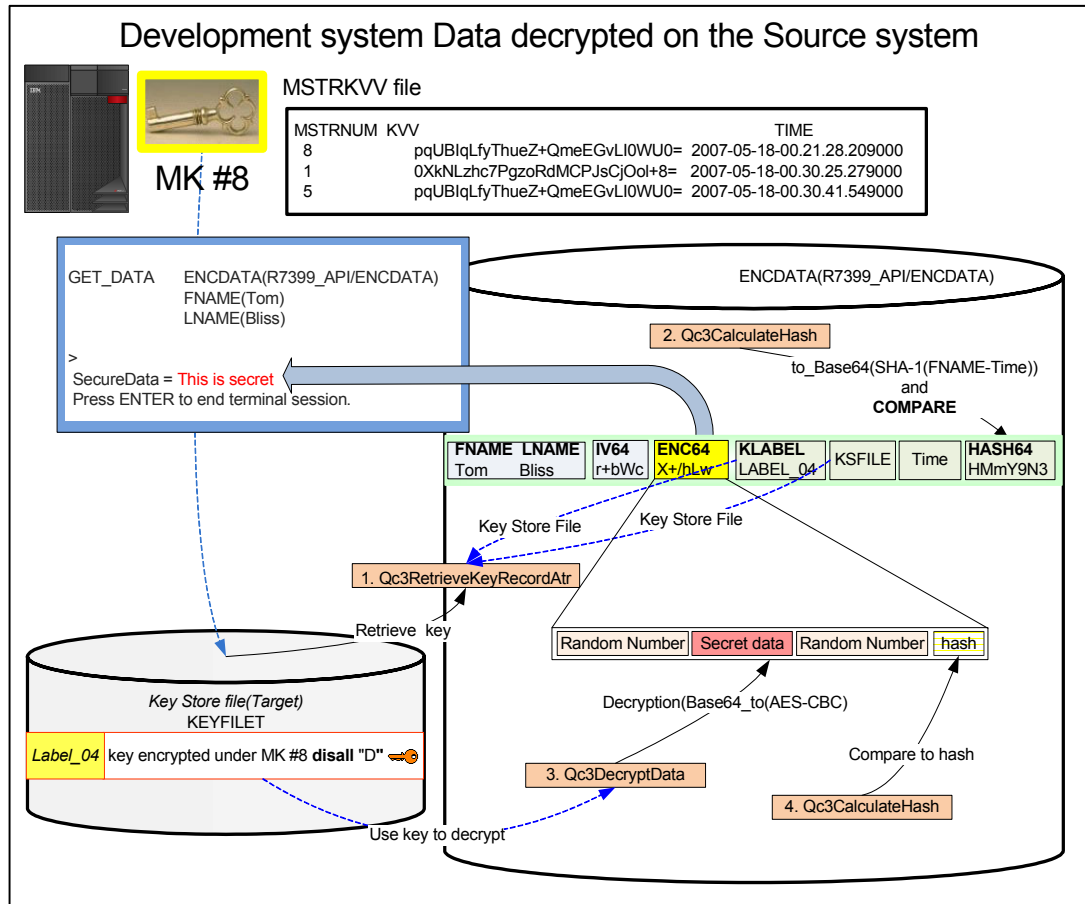


Figure 11-30 Decryption scenario

### 11.3.5 Decrypt data on target system

We assume that the customer data in file ENCDATA was saved and sent to the target system through a secure channel, and then restored on the target system. The sensitive part of the customer data was encrypted, and without the corresponding keys on the target system the data is unreadable.

We have a keystore file KEYFILET with keys that only allow decryption. Although the keystore file is encrypted under the master key (MK) on the source system, we recommend that you send this file through a different secure channel.

The keystore file is encrypted under MK #8 on the source system. On the target system, the MK should be temporarily recreated using the same master key ID (#8) and with the same passphrases. To recreate the MK we use the command SET\_MSTR\_K. In our scenario we next use program TRANS\_KEY to translate keys stored in the specified keystore files KEYFILET to a new target MK number, #1. We do this because we do not want the source system to know our master key.

If the customer does not want to recreate the MK and translate the keys in the keystore file on the target system, we can perform recreation and translation on the source system. We recreate the MK with the same number and with the same passphrase as on the target



system. We use the command TRANS\_KEY, and then we only save and restore data file ENCDATA and keystore file KEYFILET.

Now we have accessible keys in the keystore file KEYFILET encrypted under the target system MK. And now we can use the same command GET\_DATA as we used on the source system for retrieving the customer’s secret information.

### Creating temporary master key on target system

The SET\_MSTR\_K command controls the recreation of the master key for use in our application on the target system. We can use this command to create the target MK with number #1 if we have not already done so. We also use the SET\_MSTR\_K command to recreate MK #8 on the target system.

Enter the following command on the i5/OS command entry line:

```
SET_MSTR_K
```

Then press F4 for a prompt, as shown in Figure 11-31.

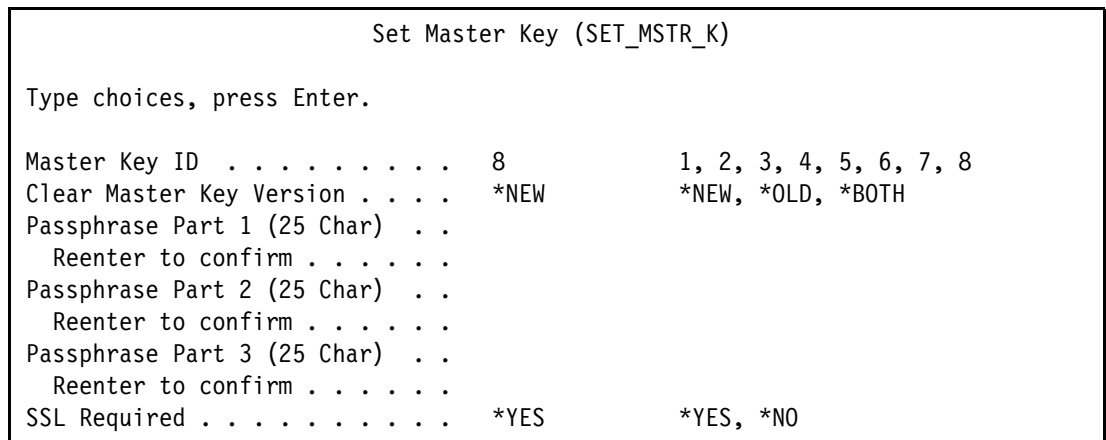


Figure 11-31 Recreate master key on the target system

For detailed description of the SET\_MSTR\_K command, refer to “Understanding SET\_MSTR\_K command” on page 145.

### Key translation

Next we execute the Translate Key Store (TRANS\_KEY) command. This command translates keys stored in the specified keystore file KEYFILET to another master key, and appears as in Figure 11-32.

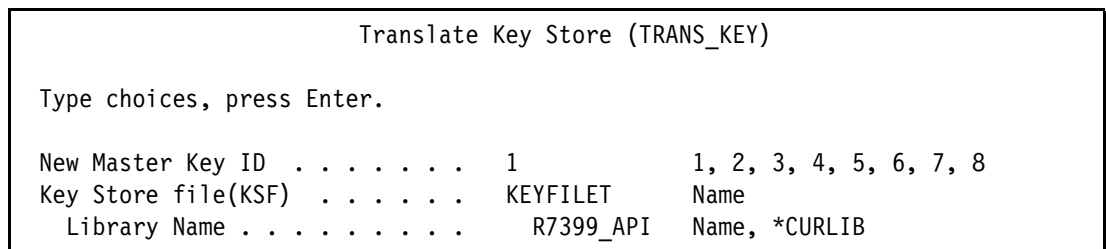


Figure 11-32 Translate keys stored in the keystore file KEYFILET to the MK #1

This command uses the following API.

## **Translate Key Store API**

This API is *Qc3TranslateKeyStore* in ILE and *QC3TRNKS* in OPM.

The Translate Key Store API translates keys stored in the specified keystore files to another master key, or if the same master key is specified, to the current version of the master key. We translate keys stored in the keystore file KEYFILEK to the target MK number #1 (Example 11-36).

### *Example 11-36 Translate Key Store API*

---

```
***argv;
switch ( *argv[0] ){
  case '1' :
    mkid = Qc3_Master_Key_1;
    break;
  case '2' :
    mkid = Qc3_Master_Key_2;
    break;
  case '3' :
    mkid = Qc3_Master_Key_3;
    break;
  case '4' :
    mkid = Qc3_Master_Key_4;
    break;
  case '5' :
    mkid = Qc3_Master_Key_5;
    break;
  case '6' :
    mkid = Qc3_Master_Key_6;
    break;
  case '7' :
    mkid = Qc3_Master_Key_7;
    break;
  default:
    mkid = Qc3_Master_Key_8;
  break; }

***argv;
memcpy( kstore.key_store, argv[0], 20);
kstore.number = 1;

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Available = sizeof(errCode);

Qc3TranslateKeyStore (&kstore,
                      &mkid,
                      &errCode);

if ( errCode.Bytes_Available != 0 )
{
  printf (">>Qc3TranslateKeyStore< \n");
  printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
}

```

---

The parameter list includes:

- ▶ kstore  
A structure containing the number of keystore files to encrypt followed by a list of keystore file names.
- ▶ mkid  
The master key under which the keys will be re-encrypted.
- ▶ errCode  
Error code structure (output parameter). This is a standard API return parameter. For more information about working with API error handling, refer to the IBM Information Center Web site:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp>

Then follow **Programming** → **Application programming interfaces** → **API concepts** → **API parameters** → **Error code parameter** to reach the page.

### Decrypting data on target system: GET\_DATA command

So far, we have translated keys stored in the keystore files KEYFILET to the target MK #1, and now we can use the same command GET\_DATA as we use on the source system for retrieving secret customer information (Figure 11-33).

```
Get Data (GET_DATA)

Data File (DF) . . . . . ENCDATA      Name
  Library Name . . . . . R7399_API    Name, *CURLIB
First Name (FN) . . . . . > Tom
Last Name (LN) . . . . . > Bliss
```

Figure 11-33 Retrieve secret data on the target system

The steps are outlined in Figure 11-34.

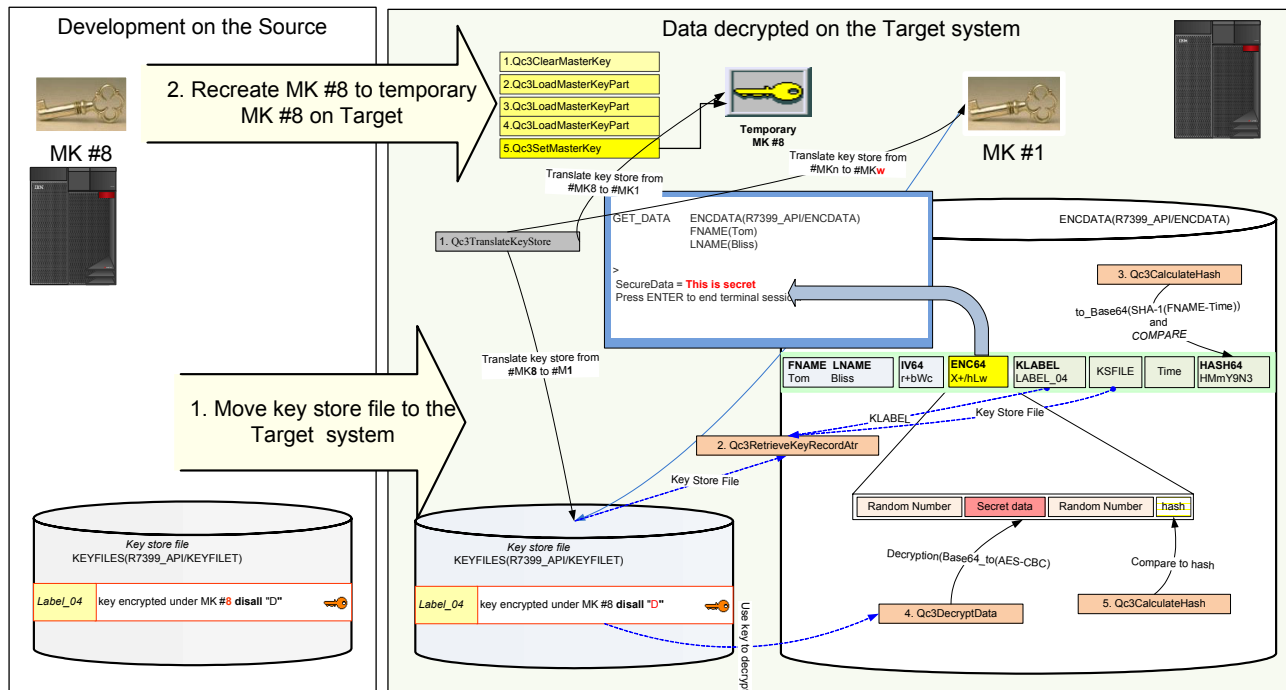


Figure 11-34 Scenario decryption data on the target system

### 11.3.6 Execution example of scenario application

This section provides an execution example of a scenario application. This might sound redundant, and it is in a way, but it gives you a more hands-on type of look and feel to the application along with actual messages that you might see. Consider this as a summary of an application description with tagged along messages.

All programs and their source code are stored in the library R7399\_API.

**Note:** Before you try to follow the instructions in this chapter, be sure that you are not on the production i5/OS server or on the system where the key management is created for another real purpose.

1. Restore library R7399\_CCA from the downloaded save file to the source system.

**Note:** When we restore library R7399\_API, all source code programs needed for our scenario are in this library.

2. Add the library R7399\_API to the user portion of the library list for the job:

```
ADDLIB LIB(R7399_API)
```

3. Compile the CL programs CRT\_MK, CRT\_SK, CRT\_SET, and CRT\_GET on the source system:

```
CRTBDCL PGM(R7399_API/CRT_MK) SRCFILE(R7399_API/QCLSRC)
        SRCMBR(CRT_MK) TEXT(*SRCMBRTXT) REPLACE(*YES) TGTRLS(*PRV)
```

```
CRTBDCL PGM(R7399_API/CRT_SK) SRCFILE(R7399_API/QCLSRC)
```

```

SRCMBR(CRT_SK) TEXT(*SRCMBRTXT) REPLACE(*YES) TGTRLS(*PRV)

CRTBNDCL PGM(R7399_API/CRT_SET) SRCFILE(R7399_API/QCLSRC)
SRCMBR(CRT_SET) TEXT(*SRCMBRTXT) REPLACE(*YES) TGTRLS(*PRV)

CRTBNDCL PGM(R7399_API/CRT_GET) SRCFILE(R7399_API/QCLSRC)
SRCMBR(CRT_GET) TEXT(*SRCMBRTXT) REPLACE(*YES) TGTRLS(*PRV)

```

4. Compile the CL programs CRT\_MK, CRT\_GET and CRT\_TK on the target system.

```

CRTBNDCL PGM(R7399_API/CRT_MK) SRCFILE(R7399_API/QCLSRC)
SRCMBR(CRT_MK) TEXT(*SRCMBRTXT) REPLACE(*YES) TGTRLS(*PRV)

CRTBNDCL PGM(R7399_API/CRT_GET) SRCFILE(R7399_API/QCLSRC)
SRCMBR(CRT_GET) TEXT(*SRCMBRTXT) REPLACE(*YES) TGTRLS(*PRV)

CRTBNDCL PGM(R7399_API/CRT_TK) SRCFILE(R7399_API/QCLSRC)
SRCMBR(CRT_TK) TEXT(*SRCMBRTXT) REPLACE(*YES) TGTRLS(*PRV)

```

5. To create all programs for our scenario on the source system, start the programs:

```

CALL CRT_MK
CALL CRT_SK
CALL CRT_SET
CALL CRT_GET

```

6. To create all programs for our scenario on the target system, start the programs:

```

CALL CRT_MK
CALL CRT_TK
CALL CRT_GET

```

7. On the source system use the SET\_MSTR\_K CL command to create Master Key #8 and enter secret strings for the three parts of the passphrase (<pass1a>, <pass2a>, <pass3a>) for MK #8.

```

SET_MSTR_K MSTRD(8) P1_1(<pass1a>) P1_2(<pass1a>)
                    P2_1(<pass2a>) P2_2(<pass2a>)
                    P3_1(<pass3a>) P3_2(<pass3a>)
                    SSL(*YES)

```

Base64(SHA-1(KVV)(28) = pqUBIqLfyThueZ+QmeEGvLI0WU0=  
Press ENTER to end terminal session.

8. Run Query to see information about generated MK:

```

RUNQRY QRY(*NONE) QRYFILE((R7399_API/MSTRKVV))

```

```

Display Report
Report width . . . . . :
Position to line . . . . . Shift to column . . . . .
Line  ....+....1....+....2....+....3....+....4....+....5....+....6....+
MSTRNUM  KVV                                TIME
000001    8    pqUBIqLfyThueZ+QmeEGvLI0WU0= 2008-02-25-13.35.46.941000
***** ***** End of report *****

```

Figure 11-35 RUNQRY QRY(\*NONE) QRYFILE((R7399\_API/MSTRKVV))

9. Generate two identical symmetric keys with different disallowed functions. Store the first key with disallowed function "0" to the keystore file KEYFILES and the second key with disallowed function D to the keystore file KEYFILET.

```
GEN_SYMKEY KEYLABEL(LABEL_04) MSTRK(8)
```

10. Run Query to see information about the generated keys in our scenario:

```
RUNQRY QRY(*NONE) QRYFILE((R7399_API/INFOFILE))
```

Position to line . . . . .	Shift to column . . . . .				
Line					
.....1.....2.....3.....4.....5.....6.....7					
	KEYLABEL	KEYTYPE	KEYSIZE	SMK	SDISALL
000001	LABEL_04	AES-CBC	16	8	0
*****	*****	End of report	*****		

Figure 11-36 RUNQRY QRY(\*NONE) QRYFILE((R7399\_API/INFOFILE))

11. Write data using command SET\_DATA (including private information) to the database file ENCDATA on the source system:

```
SET_DATA KEYLABEL(LABEL_04)
         FNAME(Tom)
         LNAME(Bliss)
         ADRES('1232 Maple St.')
         CITY(Oronoko)
         COUNTRY(USA)
         STATE(MN)
         ZIP(12346)
         PHONE(23453)
         SSN('This is secret')
```

Key Label LABEL\_04 in R7399\_API/KEYFILES was used to encrypt the data

12. Run Query to see customer information written in the database file ENCDATA in the previous step:

```
RUNQRY QRY(*NONE) QRYFILE((R7399_API/ENCDATA))
```

Position to line . . . . .	Shift to column . . . . .				
Line					
.....1.....2.....3.....4.....5.....6.....					
	FNAME	LNAME	ADRES	CITY	
000001	Tom	Bliss	1232 Maple St.	Oronoko	
*****	*****	End of report	*****		

Figure 11-37 RUNQRY QRY(\*NONE) QRYFILE((R7399\_API/ENCDATA))

13. To test whether the information was stored in database file ENCDATA in good order on the source system we use the CL command GET\_DATA:

```
GET_DATA FNAME(Tom)
         LNAME(Bliss)
```

SecureData = This is secret  
Press ENTER to end terminal session.

14. Send the data file ENCDATA with customer data and keystore file KEYFILET to the target system.

15. On the target system recreate MK #8 as a temporary MK. For this purpose use CL command SET\_STR\_K and enter the same secret strings for three parts of the passphrase (<pass1a>, <pass2a>, <pass3a>) for MK #8 as in step 7 on page 199:

```
SET_MSTR_K MSTRD(8) P1_1(<pass1a>) P1_2(<pass1a>)
                    P2_1(<pass2a>) P2_2(<pass2a>)
                    P3_1(<pass3a>) P3_2(<pass3a>)
                    SSL(*YES)
```

Base64(SHA-1(KVV))(28) = pqUBIqLfyThueZ+QmeEGvLI0WU0=  
Press ENTER to end terminal session.

16. On the target system create MK #1. For this purpose use CL command SET\_STR\_K and enter a secret string for three parts of the passphrase (<pass1b>, <pass2b>, <pass3b>) for MK #1.

```
SET_MSTR_K MSTRD(1) P1_1(<pass1b>) P1_2(<pass1b>)
                    P2_1(<pass2b>) P2_2(<pass2b>)
                    P3_1(<pass3b>) P3_2(<pass3b>)
                    SSL(*YES)
```

Base64(SHA-1(KVV))(28) = dE01mzHa0fpNf07qI/RR/n0imag=  
Press ENTER to end terminal session.

17. To translate keys stored in the keystore file ENCDATA to the MK #1 we utilize the following CL command TRANS\_KEY:

```
TRANS_KEY MSTRD(1) KEYSTORE(R7399_API/KEYFILET)
```

Key Store file re-encrypted under Master #1  
Press ENTER to end terminal session.

18. On the target system we use CL command GET\_DATA to see the secret customer information:

```
GET_DATA FNAME(Tom)
          LNAME(Bliss)
```

SecureData = This is secret  
Press ENTER to end terminal session.

## 11.4 Another scenario: for external UDFs functions

In our previous scenario, we created database file ENCDATA and put in some customer's information containing secret data. Each customer has one record, and its structure looks like Figure 11-38. The DDS source code of the database file ENCDATA is shown in Figure 11-39 on page 203, and the FNAME and LNAME are used as the database key fields.

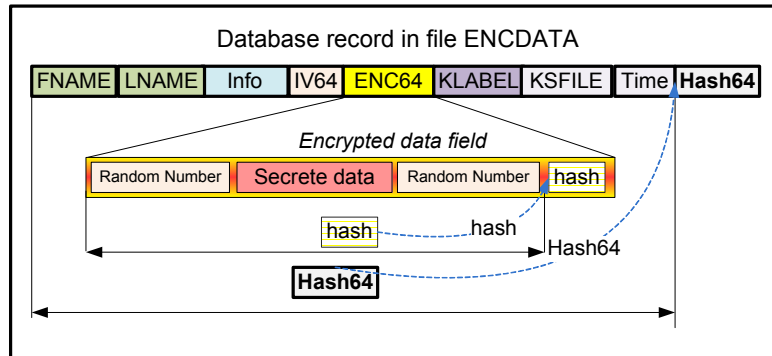


Figure 11-38 Structure of database record in the data base ENCDATA

We put in a customer's information by using CL command SET\_DATA, for example:

```
SET_DATA KEYLABEL(LABEL_04)
KEYFILES(R7399_API/KEYFILES)
KEYFILET(R7399_API/KEYFILET)
ENCDATA(R7399_API/ENCDATA)
INFODATA(R7399_API/INFOFILE)
FNAME(Tom) LNAME(Bliss)
ADRES('1232 Maple St.')
CITY('Anytown')
COUNTRY('USA')
STATE(MN) ZIP(16100)
PHONE(11111)
SSN(123123123123)
```



```

A*
A* QKSFN    Qualified Key Store File Name
A* IV      Initialization Vector
A* ENC64   Secure Data
A* PBS     Pseudorandom Binary Stream
A*
A          R ENCDATAR                                TEXT('Customer Record  ')
A*
A          FNAME      15A      TEXT('First Name      ')
A          LNAME      15A      TEXT('Last Name       ')
A*
A          ADRES      20A      TEXT(' Address        ')
A          CITY       20A      TEXT(' City            ')
A          COUNTRY    20A      TEXT(' Country         ')
A          STATE      2A       TEXT(' States          ')
A          ZIP        5A       TEXT(' ZIP Code         ')
A          PHONE      5A       TEXT(' Phone number     ')
A*
A          IV64       24A      TEXT('IV - Base64      ')
A                                     CCSID(65535)
A* Encrypted data converted to Base64
A*
A          ENC64      108A     TEXT(' Secure field    ')
A*                                     secure data
A*                                     hash
A* - end of encrypted DATA -
A*
A          KEYLAB     32A      TEXT('Key Label        ')
A          QKSFN     20A      TEXT('Key Store File Name')
A          TIME      26A      TEXT('yyyymmddhhmmss')
A          HASH64    28A      TEXT('HASH SHA-1 Base64 ')
A*
A          K FNAME
A          K LNAME
A*

```

Figure 11-39 DDS file for database file ENCDATA

To see information and list of the current customers in the database file ENCDATA, we use the following CL command:

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/ENCDATA)
```

	FNAME	LNAME	ADRES	CITY
000001	Tom	Bliss	1232 Maple St.	Oronoko
000002	Adela	Roubicek	Rumunska 88	Praha
000003	Pavel	Kalhous	U stare lipy 8	Chrudim
000004	Robin	Tatam	U nahonu239	Chariton
000005	Eva	Mandys	Kolodeje 29	Sezemice

Figure 11-40 RUNQRY QRY(\*NONE)QRYFILE(R7399\_API/ENCDATA)

## 11.4.1 External UDFs functions scenario overview

In our new scenario we want to ensure that the information in the database file ENCDATA was not inadvertently modified. That means that we want to check whether all the records written to our database file have been changed. For this purpose, each record has its proper hash value. This value is used as an integrity check value to identify this record or verify its integrity.

To illustrate how we can use the Cryptographic Services API for this purpose, we created the following two external UDF functions:

► **HASH\_DATA** function

The HASH\_DATA function checks whether the record in the ENCDATA has the correct hash. This hash is in the last column (field) in the record of the database file ENCDATA (Hash64). This field is written in the Base64 format.

<http://en.wikipedia.org/wiki/Base64>

► **DEC\_DATA** function

The DEC\_DATA function checks that the *hints* in the encrypted data field, 'hash' of the encrypted data field, and 'hash' of the entire record (Hash64) are correct. See Figure 11-38 on page 202.

For more information about UDFs function, refer to the IBM Redbooks publication, *Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries*, SG24-6503, at:

<http://www.redbooks.ibm.com/abstracts/sg246503.html?Open>

## 11.4.2 HASH\_DATA UDF function

The UDF function HASH\_DATA checks the hash of the record (Figure 11-41).

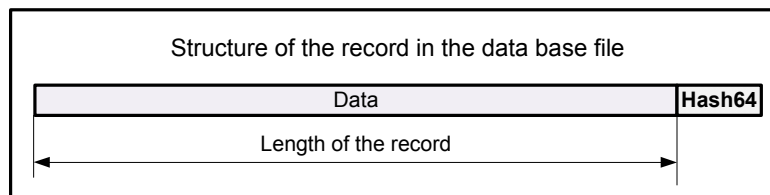


Figure 11-41 Structure of the record

### **Prerequisites and assumptions**

To prepare the scenario we must perform the following steps:

1. Create C service program TXT\_UDF\_HS:

```
CRTCMOD MODULE(R7399_API/TXT_UDF_HS)
        SRCFILE(R7399_API/QCSRC_SQL)
        TEXT('SQL UFD function to Test Hash of rows ')

CRTSRVPGM SRVPGM(R7399_API/TXT_UDF_HS)
        MODULE(R7399_API/TXT_UDF_HS R7399_API/BASE64_COD)
        EXPORT(*ALL) ACTGRP(*CALLER)
```

## 2. Create UDF function HASH\_DATA.

Before a UDF function can be recognized and used by the database manager, it should be created using the CREATE FUNCTION statement (Figure 11-42).

STRSQL

```
Type SQL statement, press Enter..
> CREATE FUNCTION R7399_API/HASH_DATA(VARCHAR(312), INTEGER,
    VARCHAR(108))
    RETURNS      VARCHAR(3)
    LANGUAGE     C
    DETERMINISTIC
    specific     HASHED0001
    NO SQL
    NO EXTERNAL ACTION
    external name 'R7399_API/TXT_UDF_HS(txt_UDF_hash)'
    parameter style SQL
Function HASH_DATA was created in R7399_API.
```

Figure 11-42 Create SQL UDF function HASH\_DATA

This function utilizes the Calculate HASH API, which calculates a HASH value for a string. For further details of this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/index.jsp?topic=/apis/qc3calha.htm>

### **HASH\_DATA**

To calculate and check the hash of the database record in file ENCDATA, we enter the CL command STRSQL and then the SQL statement (Figure 11-43).

STRSQL

```
SELECT FNAME, LNAME,
    HASH_DATA( FNAME || LNAME || ADRES || CITY ||
    COUNTRY || STATE || ZIP || PHONE ||
    IV64 || ENC64 || KEYLAB || QKSFN ||
    TIME ,
    char_length( FNAME || LNAME || ADRES || CITY
    ||
    COUNTRY || STATE || ZIP || PHONE
    ||
    IV64 || ENC64 || KEYLAB || QKSFN
    ||
    TIME ),
    HASH64 ) as Result
from ENCDATA
```

Figure 11-43 SQL statement to use UDF function HASH\_DATA

The external UDF function, HASH\_DATA, calculates the hash of the string and compares it with the expected value.

It requires the following three input parameters:

- The data string that will be hashed:

```
FNAME || LNAME || ADRES || CITY ||
COUNTRY || STATE || ZIP || PHONE ||
```

```
IV64    || ENC64 || KEYLAB || QKSFN ||
TIME
```

- Length of the data string:

```
char_length( FNAME    || LNAME || ADRES  || CITY
             ||
             COUNTRY || STATE || ZIP    || PHONE
             ||
             IV64     || ENC64 || KEYLAB || QKSFN
             ||
             TIME )
```

- Expected hash value (in our scenario the last field of the record (Hash64)):

```
HASH64
```

The output parameter tells you whether the hash of the string and the expected hash value are the same. The value of this output parameter is YES or NO (Figure 11-44 column RESULT).

FNAME	LNAME	RESULT
Tom	Bliss	YES
Adeła	Roubicek	YES
Pavel	Kalhous	YES
Robin	Tatam	YES
Eva	Mandys	YES

Figure 11-44 Result of the SQL statement

The source code for the txt\_UDF\_hash function is shown in Example 11-37.

Example 11-37 C source code for the txt\_UDF\_hash function

```
#include <recio.h> /* Record I/O routines */
#include <qusec.h> /* Error code structure */
#include <sqludf.h>
#include <qc3ctx.h> /* Hdr for Context APIs */

int to_Base64( char *a, int b, char *c, int d );

void SQL_API_FN txt_UDF_hash ( char * Concatenated_string,
                              int * Length_Concatenated_string,
                              char * Hash_Base64,

                              SQLUDF_CHAR *rtn_Yes_or_No,

                              SQLUDF_NULLIND *nms_InputNullIndicator01,
                              SQLUDF_NULLIND *nms_OutputNullIndicator01,
                              SQLUDF_CHAR sqludf_sqlstate[ SQLUDF_SQLSTATE_LEN + 1 ],
                              SQLUDF_CHAR sqludf_fname[ SQLUDF_FQNAME_LEN + 1 ],
                              SQLUDF_CHAR sqludf_specname[ SQLUDF_SPECNAME_LEN + 1 ],
                              SQLUDF_CHAR sqludf_msgtext[ SQLUDF_MSGTEXT_LEN + 1 ] )
{
    Qus_EC_t errCode; /* Error code structure */

    char data_field[999];
    char hash_string[20];
```

```

char      hash_string_Base64[28];
int       Len_data;
char      Algorithm_description[04];
char Cryptographic_service_provider;

memset ( Algorithm_description, 0, sizeof(Algorithm_description));
Algorithm_description[3] = Qc3_SHA1;
Cryptographic_service_provider = Qc3_Any_CSP;

Len_data = *Length_Concatenated_string;
memcpy( data_field, Concatenated_string, Len_data );

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

Qc3CalculateHash (data_field,
                  &Len_data,
                  "DATA0100",
                  (char*)&Algorithm_description,
                  "ALGD0500",
                  &Cryptographic_service_provider,
                  NULL,
                  hash_string,
                  &errCode );

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3CalculateHash \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return ;
}
Len_data = to_Base64( hash_string, sizeof(hash_string),
                    hash_string_Base64, Len_data );

if ( memcmp( Hash_Base64, hash_string_Base64,
            sizeof(hash_string_Base64)) != 0 )
    memcpy ( rtn_Yes_or_No, "NO ", 3 );
else
    memcpy ( rtn_Yes_or_No, "YES", 3 );

return;
}

```

---

### 11.4.3 DEC\_DATA UDF function

Now we create new UDF function DEC\_DATA and add more input parameters to be able to check the encrypted data field and check the hash of the entire record (Figure 11-38 on page 202).

## Prerequisites and assumptions

To prepare the scenario we must perform the following steps:

1. Create C service program TXT\_UDF\_AL:

```
CRTCMOD MODULE(R7399_API/GET_UDF_AL)
        SRCFILE(R7399_API/QCSRC_SQL)
        TEXT('SQL UFD function to GET encrypted data')
CRTSRVPGM SRVPGM(R7399_API/GET_UDF_AL)
        MODULE( R7399_API/GET_UDF_AL R7399_API/BASE64_COD)
        EXPORT(*ALL) ACTGRP(*CALLER)
```

2. Create UDF function DEC\_DATA.

Before a UDF function can be recognized and used by the database manager, it should be created using the CREATE FUNCTION statement (Figure 11-45):

STRSQL

```
CREATE FUNCTION R7399_API/DEC_DATA(VARCHAR(32), VARCHAR(20),
        VARCHAR(28), VARCHAR(108),
        VARCHAR(312), INTEGER, VARCHAR(28))
        RETURNS          VARCHAR(14)
        LANGUAGE          C
        DETERMINISTIC
        specific          DECRY0003
        NO SQL
        NO EXTERNAL ACTION
        external name 'R7399_API/GET_UDF_AL(get_UDF_A11)'
        parameter style  SQL
```

Figure 11-45 Create SQL UDF function DEC\_DATA

This UDF functions use the following APIs:

- ▶ Calculate Hash API: This API calculates a hash value for a string. For further details on this API, visit the following link:  
<http://publib.boulder.ibm.com/infocenter/iseriess/v5r3/index.jsp?topic=/apis/qc3calha.htm>
- ▶ Decrypt Data API: This API restores encrypted data to a clear (intelligible) form. For further details on this API, visit the following link:  
<http://publib.boulder.ibm.com/iseriess/v5r2/ic2924/index.htm?info/apis/qc3decdt.htm>

## 11.4.4 Running DEC\_DATA command

To calculate and check the hash of the database record in file ENCDATA and calculate and check the hash of the encrypted data field, we enter the CL command STRSQL and then the SQL statement (Figure 11-46).

```
select FNAME, LNAME,
       DEC_DATA(KEYLAB, QKSFN, IV64,
               ENC64,
               FNAME || LNAME || ADRES || CITY ||
               COUNTRY || STATE || ZIP || PHONE ||
               IV64 || ENC64 || KEYLAB || QKSFN ||
               TIME,
               char_length( FNAME || LNAME || ADRES || CITY ||
                           COUNTRY || STATE || ZIP || PHONE ||
                           IV64 || ENC64 || KEYLAB || QKSFN ||
                           TIME ),
               HASH64 ) as RESULT
from ENCDATA
```

Figure 11-46 SQL statement to use UDF function DEC\_DATA

The external UDF function DEC\_DATA calculates the hash of the entire record and compares it with the written hash value (Hash64). It then decrypts the encrypted data field, calculates the hash, and compares it with the written hash (hash). It requires the following seven input parameters:

- ▶ The key label used for encrypting the secret data in the customer data record:  
KEYLAB
- ▶ The fully qualified file name of the keystore file containing the keys used for encrypting the secret data:  
QKSFN
- ▶ The Initialization vector in the Base64 format:  
IV64
- ▶ The encrypted data field:  
ENC64

- ▶ The data that will be hashed:

```
FNAME || LNAME || ADRES || CITY ||
COUNTRY || STATE || ZIP || PHONE ||
IV64 || ENC64 || KEYLAB || QKSFN ||
TIME
```

- ▶ The length of the data string:

```
char_length( FNAME || LNAME || ADRES || CITY
            ||
            COUNTRY || STATE || ZIP || PHONE
            ||
            IV64 || ENC64 || KEYLAB || QKSFN
            ||
            TIME )
```

- The expected hash value (in our scenario the last field of the record (Hash64)):

HASH64

The output parameter tells you whether the data in the record was modified. If the data in the record was not changed, we receive as a result the secret data (Figure 11-47).

FNAME	LNAME	RESULT
Tom	Bliss	123123123123
Adela	Roubicek	>>Error HASH<<
Pavel	Kalhous	Ahoj Pavle jak
Robin	Tatam	Hi Robin Tatam
Eva	Mandys	This This This

Figure 11-47 Result of SQL statement of DEC\_DATA

For the customer Adela Roubicek we can see as a result >>Error HASH<<. Some part of the record was changed. For the rest of the customers we can see their secret data.

Example 11-38 Example of C source code GET\_UDF\_AL UDF function

```
#include <recio.h> /* Record I/O routines */
#include <qusec.h> /* Error code structure */
#include <sqludf.h>
#include <qc3ctx.h> /* Hdr for Context APIs */

#define len_PRBS 16 /* Pseudorandom Binary Stream */
#define len_Sec 14 /* Length Secret Data 14Byte */
#define len_FNAME 15 /* length of First Name */
#define len_LNAME 15 /* length of Last Name */

int Base64_to( char *a, int b, char *c, int d );

void SQL_API_FN get_UDF_All ( char * Key_Label,
                             char * Qualified_key_store_file_name,
                             char * IV_B64,
                             char * Encrypted_64_Data,
                             char * Concatenated_string,
                             int * Length_Concatenated_string,
                             char * Hash_Base64,

                             SQLUDF_CHAR *rtn_Decrypted_Secret_Data,

                             SQLUDF_NULLIND *nms_InputNullIndicator01,
                             SQLUDF_NULLIND *nms_OutputNullIndicator01,
                             SQLUDF_CHAR sqludf_sqlstate[ SQLUDF_SQLSTATE_LEN + 1 ],
                             SQLUDF_CHAR sqludf_fname[ SQLUDF_FQNAME_LEN + 1 ],
                             SQLUDF_CHAR sqludf_specname[ SQLUDF_SPECNAME_LEN + 1 ],
                             SQLUDF_CHAR sqludf_msgtext[ SQLUDF_MSGTEXT_LEN + 1 ] )
{
    Qc3_Format_ALGD0200_T algD; /* Block cipher alg description*/
    Qc3_Format_KEYD0400_T kskey; /* Key store key structure */
    Qus_EC_t errorCode; /* Error code structure */

    char IV[16];
```



```

char      Data_enc[80];
char      n_Data_enc[80];
char      HASH[20];
int       rtnLen;
int       len_in;
int       len_out;
int       cipherLen;
int       i;

char      data_field[999];
char      hash_string[20];
char      hash_string_Base64[28];
int       Len_data;
char      Algorithm_description[04];
char      Cryptographic_service_provider;

/*-----*/
/* 1. Calculate hash SHA - 1 for all record */
/*-----*/
Cryptographic_service_provider = Qc3_Any_CSP;
memset ( Algorithm_description, 0, sizeof(Algorithm_description));
Algorithm_description[3] = Qc3_SHA1;
Len_data = *Length_Concatenated_string;
memcpy( data_field, Concatenated_string, Len_data );

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

Qc3CalculateHash ((char*)&data_field,
                  &Len_data,
                  "DATA0100",
                  (char*)&Algorithm_description,
                  "ALGD0500",
                  &Cryptographic_service_provider,
                  NULL,
                  hash_string,
                  &errCode );

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3CalculateHash \n");
    printf ( "  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return ;
}
Len_data = to_Base64( hash_string, sizeof(hash_string),
                     hash_string_Base64, Len_data );
if ( memcmp( Hash_Base64, hash_string_Base64,
             sizeof(hash_string_Base64)) != 0 )
{
    printf (">>Error HASH << \n");
    memset ( rtn_Decrypted_Secrete_Data, ' ',
            sizeof(rtn_Decrypted_Secrete_Data));
    memcpy ( rtn_Decrypted_Secrete_Data, ">>Error HASH<<", len_Sec);
    return;
}

```

```

/*-----*/
/* 1. Convert IV and ENCDATA from Base64 to Binary Stream */
/*-----*/
memset(&algD, 0, sizeof(algD)); /* Init alg description to null*/
len_in = 24;
len_out = Base64_to( IV_B64, len_in, algD.Init_Vector, len_out);
len_in = 108;
len_out = Base64_to( Encrypted_64_Data, len_in,
                    Data_enc, len_out);

/*-----*/
/* 2. Create Algorithm description for Format Name "ALGD0200" */
/*-----*/
algD.Block_Cipher_Alg = Qc3_AES; /* Set AES algorithm */
algD.Block_Length = 16; /* Block size is 16 */
algD.Mode = Qc3_CBC; /* Use cipher block chaining */
algD.Pad_Option = Qc3_No_Pad; /* Do not pad */

/*-----*/
/* 3. Setup Key description for format name "KEYD0400" */
/*-----*/
memset(&kskey, '\0', sizeof(kskey));
memcpy( kskey.Key_Store, Qualified_key_store_file_name, 20 );
memcpy( kskey.Record_Label, Key_Label, 32 );
cipherLen = len_out;

/*-----*/
/* 4. Decrypt the Data */
/*-----*/
memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

Qc3DecryptData( Data_enc,
                &len_out,
                (char*)&algD,
                Qc3_Alg_Block_Cipher,
                &kskey,
                Qc3_Key_KSLabel,
                &Cryptographic_service_provider,
                NULL,
                Data_enc,
                &cipherLen,
                &rtLen,
                &errCode);

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3EncryptData \n");
    printf (" errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    return ;
}

/*-----*/
/* 5. Calculate SHA-1 hash for secure data */
/*-----*/
len_out = 60;
memset ( Algorithm_description, 0, sizeof(Algorithm_description));
Algorithm_description[3] = Qc3_SHA1;

```

```

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

Qc3CalculateHash (Data_enc,
                  &len_out,
                  "DATA0100",
                  (char*)&Algorithm_description,
                  "ALGD0500",
                  &Cryptographic_service_provider,
                  NULL,
                  HASH,
                  &errCode );

if ( errCode.Bytes_Available != 0 ) {
    printf (">>Qc3CalculateHash \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    memset ( rtn_Decrypted_Secret_Data, ' ', len_Sec);
    memcpy ( rtn_Decrypted_Secret_Data, ">>Error Qc3h<<", len_Sec);
    return;
}
if ( memcmp(&Data_enc[60], HASH, 20) != 0 ) {
    printf (">>HASH error secret data \n");
    memset ( rtn_Decrypted_Secret_Data, ' ', len_Sec);
    memcpy ( rtn_Decrypted_Secret_Data, ">>Error hash<<", len_Sec);
    return;
}
memset ( rtn_Decrypted_Secret_Data, ' ', len_Sec);
memcpy ( rtn_Decrypted_Secret_Data, Data_enc+len_PRBS, len_Sec );
printf ("SecureData = %.14s\n", rtn_Decrypted_Secret_Data);

return;
}

```

---

### 11.4.5 Execution example of external UDFs function scenario

This section provides an execution example of our UDF's function scenario. This might sound redundant, and it is in a way, but it gives you a more hands-on look and feel of the application along with actual messages that you might see. Consider this as a summary of an application description with tagged along messages.

All source code is stored in the library R7399\_API in the source physical file QCSRC\_SQL.

Before we start this section we assume the library R7399\_API is restored, the MK is set up, and at least one symmetric key is in the keystore file KEYFILET encrypted under this MK.

1. Run Query to see information about generated MKs (Figure 11-48):

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/MSTRKVV)
```

	MSTRNUM	KVV	TIME
000001	8	pqUBIqLfYThueZ+QmeEGvLIOWU0=	2008-02-26-14.01.18.080000
000002	1	dE01mzHa0fpNf07qI/RR/n0imag=	2008-02-26-14.01.29.412000

Figure 11-48 List of MK

- Run Query to see information about the generated symmetric keys (Figure 11-49):

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/INFOFILE)
```

KEYLABEL	KEYTYPE	KEYSIZE	SMK
000001 LABEL_04	AES-CBC	16	8
000002 LABEL_01	AES-CBC	16	8
000003 LABEL_08	AES-CBC	16	8

Figure 11-49 List of generated symmetric keys

- Add the library R7399\_API to the user portion of the library list for the job:

```
ADDLIB LIB(R7399_API)
```

- Create C service program txt\_UDF\_HS:

```
CRTCMOD MODULE(R7399_API/TXT_UDF_HS)
  SRCFILE(R7399_API/QCSRC_SQL)
  TEXT('SQL UFD function to Test Hash of rows ')

CRTSRVPGM SRVPGM(R7399_API/TXT_UDF_HS)
  MODULE(R7399_API/TXT_UDF_HS R7399_API/BASE64_COD)
  EXPORT(*ALL) ACTGRP(*CALLER)
```

- Create UDF function HASH\_DATA:

```
STRSQL

CREATE FUNCTION R7399_API/HASH_DATA(VARCHAR(312), INTEGER,
  VARCHAR(108))
  RETURNS VARCHAR(3)
  LANGUAGE C
  DETERMINISTIC
  specific HASHED0001
  NO SQL
  NO EXTERNAL ACTION
  external name 'R7399_API/TXT_UDF_HS(txt_UDF_hash)'
  parameter style SQL
```

When we receive message Routine HASH\_DATA in R7399\_API already exists we can use the DROP function to remove this definition:

```
DROP FUNCTION R7399_API/HASH_DATA
```

- Create the C service program txt\_UDF\_AL:

```
CRTCMOD MODULE(R7399_API/GET_UDF_AL)
  SRCFILE(R7399_API/QCSRC_SQL)
  TEXT('SQL UFD function to GET encrypted data')

CRTSRVPGM SRVPGM(R7399_API/GET_UDF_AL)
  MODULE( R7399_API/GET_UDF_AL R7399_API/BASE64_COD)
  EXPORT(*ALL) ACTGRP(*CALLER)
```

- Create the UDF function DEC\_DATA:

```
STRSQL

CREATE FUNCTION R7399_API/DEC_DATA(VARCHAR(32), VARCHAR(20),
  VARCHAR(28), VARCHAR(108),
  VARCHAR(312), INTEGER, VARCHAR(28))
```

```

        RETURNS          VARCHAR(14)
        LANGUAGE         C
        DETERMINISTIC
        specific        DECRYPT0003
        NO SQL
        NO EXTERNAL ACTION
        external name 'R7399_API/GET_UDF_AL(get_UDF_A11)'
        parameter style SQL

```

When we receive the message Routine DEC\_DATA in R7399\_API already exists we can use the DROP function to remove this definition:

```
DROP FUNCTION R7399_API/DEC_DATA
```

8. Create the C service program txt\_UDF\_DS:

```

CRTCMOD MODULE(R7399_API/TXT_UDF_DS)
        SRCFILE(R7399_API/QCSRC_SQL)
        TEXT('SQL UFD function to Display Hash of rows ')

CRTSRVPGM SRVPGM(R7399_API/TXT_UDF_DS)
        MODULE( R7399_API/TXT_UDF_DS R7399_API/BASE64_COD)
        EXPORT(*ALL) ACTGRP(*CALLER)

```

9. Create UDF function HASH\_DSP:

```

STRSQL

CREATE FUNCTION R7399_API/HASH_DSP(VARCHAR(312), INTEGER,
        VARCHAR(108))
        RETURNS          VARCHAR(28)
        LANGUAGE         C
        DETERMINISTIC
        specific        HASHED0002
        NO SQL
        NO EXTERNAL ACTION
        external name 'R7399_API/TXT_UDF_DS(txt_UDF_dsp)'
        parameter style SQL

```

When we receive the message Routine HASH\_DSP in R7399\_API already exists we can use the DROP function to remove this definition:

```
DROP FUNCTION R7399_API/HASH_DSP
```

10. Create the physical file ENCDATA if we have not already done so:

```

CRTPF FILE(R7399_API/ENCDATA) SRCFILE(R7399_API/QDDSSRC)
        SRCMBR(ENCDATA) GENLVL(20) FLAG(0) FILETYPE(*DATA)

```

11. Write data (including private information) to the database file ENCDATA:

```

SET_DATA KEYLABEL(LABEL_04)
        KEYFILES(R7399_API/KEYFILES)
        KEYFILET(R7399_API/KEYFILET)
        ENCDATA(R7399_API/ENCDATA)
        INFODATA(R7399_API/INFOFILE)
        FNAME(Tom) LNAME(Bliss)
        ADRES('1232 Maple St.')
        CITY('Oronoko')
        COUNTRY('USA')
        STATE(MN) ZIP(16100)
        PHONE(11111)

```

SSN(123123123123)

```
SET_DATA KEYLABEL(LABEL_01)
KEYFILES(R7399_API/KEYFILES)
KEYFILET(R7399_API/KEYFILET)
ENCADATA(R7399_API/ENCADATA)
INFODATA(R7399_API/INFOFILE)
FNAME(Adela) LNAME(Roubicek)
ADRES('Rumunska 88')
CITY(Praha)
COUNTRY('Czech Republic')
STATE(CZ)
ZIP(11022)
PHONE(26588)
SSN('Text Text Text')
```

```
SET_DATA KEYLABEL(LABEL_08)
KEYFILES(R7399_API/KEYFILES)
KEYFILET(R7399_API/KEYFILET)
ENCADATA(R7399_API/ENCADATA)
INFODATA(R7399_API/INFOFILE)
FNAME(Pavel)
LNAME(Kalhous)
ADRES('Na ruzku 8')
CITY(Chrudim)
COUNTRY('Czech Republic')
STATE(CZ) ZIP(52692)
PHONE(12698)
SSN('Ahoj Pavle jak')
```

```
SET_DATA KEYLABEL(LABEL_08)
KEYFILES(R7399_API/KEYFILES)
KEYFILET(R7399_API/KEYFILET)
ENCADATA(R7399_API/ENCADATA)
INFODATA(R7399_API/INFOFILE)
FNAME(Robin)
LNAME(Tatam)
ADRES('Country Rd 59')
CITY('Chariton')
COUNTRY('USA')
STATE(IO)
ZIP(12682)
PHONE(82345)
SSN('Hi Robin Tatam')
```

12. Run an SQL statement to use the UDF function HASH\_DATA:

```
SELECT FNAME, LNAME,
       HASH_DATA( FNAME || LNAME || ADRES || CITY ||
                 COUNTRY || STATE || ZIP || PHONE ||
                 IV64 || ENC64 || KEYLAB || QKSFN ||
                 TIME ,
                 char_length( FNAME || LNAME || ADRES || CITY
                              ||
                              COUNTRY || STATE || ZIP || PHONE
                              ||
                              )
              )
```

```

        IV64    || ENC64 || KEYLAB || QKSFN
        ||
        TIME ),
    HASH64 ) as Result
from ENCDATA

```

FNAME	LNAME	RESULT
Tom	Bliss	YES
Adela	Roubicek	YES
Pavel	Kalhous	YES
Robin	Tatam	YES

Figure 11-50 SQL statement UDF function HASH\_DATA

13. Run the SQL statement to use the UDF function HASH\_DSP:

```

SELECT FNAME, LNAME,
    HASH_DSP ( FNAME || LNAME || ADRES || CITY ||
        COUNTRY || STATE || ZIP || PHONE ||
        IV64 || ENC64 || KEYLAB || QKSFN ||
        TIME ,
        char_length( FNAME || LNAME || ADRES || CITY
            ||
            COUNTRY || STATE || ZIP || PHONE
            ||
            IV64 || ENC64 || KEYLAB || QKSFN
            ||
            TIME ),
    HASH64 ) as Result
from ENCDATA

```

FNAME	LNAME	RESULT
Tom	Bliss	/ExbvFFjtKGUGMrqKQ3XLID5JfM=
Adela	Roubicek	eShyU1xftpW8Vcw9QTyViwGn9Pis=
Pavel	Kalhous	47au5v5PqnMgnUXy0fusIeZLmW8=
Robin	Tatam	MqfoS9uM3HsYnNSRUQDuNpbqvA=

Figure 11-51 SQL statement UDF function HASH\_DATA

14. Run the SQL statement to use the UDF function HASH\_DATA:

```

SELECT FNAME, LNAME,
    (KEYLAB, QKSFN, IV64,
    ENC64,
    FNAME || LNAME || ADRES || CITY ||
    COUNTRY || STATE || ZIP || PHONE ||
    IV64 || ENC64 || KEYLAB || QKSFN ||
    TIME,
    char_length( FNAME || LNAME || ADRES || CITY ||
        COUNTRY || STATE || ZIP || PHONE ||
        IV64 || ENC64 || KEYLAB || QKSFN ||
        TIME ),
    HASH64 ) as RESULT
from ENCDATA

```

FNAME	LNAME	RESULT
Tom	Bliss	123123123123
Adela	Roubicek	Text Text Text
Pavel	Kalhous	Ahoj Pavle jak
Robin	Tatam	Hi Robin Tatam

Figure 11-52 SQL statement UDF function

## 11.4.6 Using the external trigger function

On the target system, there is another customer database file EMPDATA that includes the same information as the database file ENCDATA except for the fields needing decryption (IV64, KEYLAB, QKSFN) and fields TIME and HASH64. The structure of this record looks like Figure 11-53. The EMPDATA file is shown in Figure 11-54 on page 219.

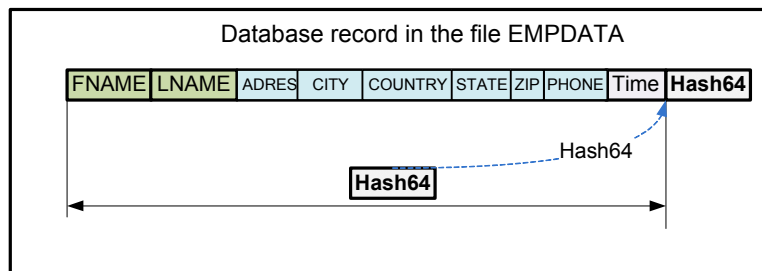


Figure 11-53 DDS source code for the database file ENCDATA

In our scenario there are some requirements. If some information has changed (for example, phone number (PHOME field), address (ADRES field)) or some records are deleted from the EMPDATA file, we must expose these changes in the ENCDATA file, and if the record is not deleted, the hash of the record should be recalculated. The FNAME and the LNAME are used as the database key fields, and therefore these two fields are unchangeable.

We created a user-written program trigger, which defines a set of required actions executed in response to the event of an UPDATE or DELETE operation on the EMPDATA file.

More information about DB2 Triggers we can find at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/ad/c0007038.htm>

Also refer to the IBM Redbooks publication, *Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries*, SG24-6503, at:

<http://www.redbooks.ibm.com/abstracts/sg246503.html?Open>



A*				
A	R	EMPDATA		TEXT('Employee Record')
A*				
A		FNAME	15A	TEXT('First Name')
A		LNAME	15A	TEXT('Last Name')
A*				
A		ADRES	20A	TEXT(' Address')
A		CITY	20A	TEXT(' City')
A		COUNTRY	20A	TEXT(' Country')
A		STATE	2A	TEXT(' States')
A		ZIP	5A	TEXT(' ZIP Code')
A		PHONE	5A	TEXT(' Phone number')
A*				
A		<b>K FNAME</b>		
A		<b>K LNAME</b>		
A*				

Figure 11-54 Database file EMPDATA

In our scenario we created external trigger program UPD\_TRG\_DT for the database file EMPDATA. This trigger program exposes some changes in the file EMPDATA immediately into the database file ENCDATAh.

**Prerequisites and assumptions**

To prepare our scenario we need to create the database file EMPDATA and perform the following steps:

1. Create the physical file EMPDATA:

```
CRTPF FILE(R7399_API/EMPDATA) SRCFILE(R7399_API/QDDSSRC)
      SRCMBR(EMPDATA) TEXT('Employee database file')
```

2. Insert into this database file EMPDATA the same information that we put into the database file ENCDATA, without the secret data. To insert information we start SQL:

STRSQL

Use SQL statements:

```
INSERT INTO R7399_API/EMPDATA
  values('Tom',
        'Bliss',
        '1232 Maple St.',
        'Oronoko',
        'USA',
        'MN',
        '16100',
        '11111')
```

3. Run Query to see a list of records put in the database file EMPDATA:

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/EMPDATA)
```

FNAME	LNAME	ADRES	CITY
000001 Robin	Tatam	Country Rd 59	Chariton
000002 Tom	Bliss	1232 Maple St.	Oronoko
000003 Adela	Roubicek	Rumunska 88	Praha
000004 Pavel	Kalhous	Na ruzku 8	Chrudim

Figure 11-55 RUNQRY QRY(\*NONE) QRYFILE(R7399\_API/EMPDATA)

4. Run Query to see a list of records put in the database file ENCDATA:

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/ENCDATA)
```

FNAME	LNAME	ADRES	CITY
000001 Tom	Bliss	1232 Maple St.	Oronoko
000002 Adela	Roubicek	Rumunska 88	Praha
000003 Pavel	Kalhous	Na ruzku 8	Chrudim
000004 Robin	Tatam	Country Rd 59	Chariton

Figure 11-56 RUNQRY QRY(\*NONE) QRYFILE(R7399\_API/ENCDATA)

5. Define an external trigger.

To define an external trigger, we must create an external trigger program and add it to a table using the Add Physical File Trigger (ADDPFTRG) CL command.

a. Create C program UPD\_TRG\_DT:

```
CRTCMOD MODULE(R7399_API/UPD_TRG_DT)
          SRCFILE(R7399_API/QCSRC_SQL)
```

```
CRTPGM PGM(R7399_API/UPD_TRG_DT)
        MODULE(R7399_API/UPD_TRG_DT
              R7399_API/BASE64_COD )
```

b. Add a trigger to a table.

To add a trigger to a table, we must:

- Identify the table:

```
R7399_API/EMPDATA
```

- Identify the kind of operation:

```
TRGTIME(*AFTER) TRGEVENT(*UPDATE), RGTIME(*BEFORE) TRGEVENT(*DELETE)
```

- Identify the program that performs the desired actions:

```
PGM(R7399_API/UPD_TRG_DT)
```

- Provide a unique name for the trigger or let the system generate a unique name:

```
TRG(UPD_TRG_DT_UPDATE), TRG(UPD_TRG_DT_DELETE)
```

Create trigger R7399\_API/UPD\_TRG\_DT to update the EMPDATA file:

```
ADDPFTRG FILE(R7399_API/EMPDATA)
          TRGTIME(*AFTER) TRGEVENT(*UPDATE)
          PGM(R7399_API/UPD_TRG_DT)
          TRG(UPD_TRG_DT_UPDATE)
          TRGLIB(R7399_API)
```

```
MLTTHDACN(*RUN) TRGUPDCND(*CHANGE)
```

Create trigger R7399\_API/UPD\_TRG\_DT to delete from the EMPDATA file:

```
ADDPFTRG FILE(R7399_API/EMPDATA)
TRGTIME(*BEFORE) TRGEVENT(*DELETE)
PGM(R7399_API/UPD_TRG_DT)
TRG(UPD_TRG_DT_DELETE)
TRGLIB(R7399_API) MLTTHDACN(*RUN)
```

6. Now we are ready to walk through our scenario.

For customer Adela Roubicek we want to change a place of residence and for customer Pavel Kalhous we want to remove his record from the database file EMPDATA. For customer Adela Roubicek our external trigger program changes the information in database file ENCDATA and recalculates the hash, and for customer Pavel Kalhous our external trigger program deletes his record from the ENCDATA file.

To update record Adela Roubickova in the EMPNAME file we use the SQL statement:

```
STRSQL
```

```
UPDATE EMPDATA
SET CITY = 'Kolodeje 29', ADRES = 'Sezemice'
WHERE FNAME = 'Adela'
```

To delete record Pavel Kalhous in the EMPNAME file we use the SQL statement:

```
DELETE FROM EMPDATA
WHERE FNAME = 'Pavel' AND LNAME = 'Kalhous'
```

7. We run Query to see a list of records in the database file ENCDATA and EMPDATA:

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/ENCDATA)
```

	FNAME	LNAME	ADRES	CITY
000001	Tom	Bliss	1232 Maple St.	Oronoko
000002	Adela	Roubicek	Sezemice	Kolodeje 29
000003	Robin	Tatam	Country Rd 59	Chariton

Figure 11-57 RUNQRY QRY(\*NONE) QRYFILE(R7399\_API/ENCDATA)

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/EMPDATA)
```

	FNAME	LNAME	ADRES	CITY
000001	Robin	Tatam	Country Rd 59	Chariton
000002	Tom	Bliss	1232 Maple St.	Oronoko
000003	Adela	Roubicek	Sezemice	Kolodeje 29

Figure 11-58 RUNQRY QRY(\*NONE) QRYFILE(R7399\_API/EMPDATA)

We use the Calculate Hash API to calculate a hash value of a string. For further details on this API, visit the following link:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/index.jsp?topic=/apis/qc3calha.htm>

## UPD\_TRG\_DT

This program is called when a delete or an update operation occurs in the EMPDATA file. This program will delete or update the records from the ENCDATA file based on the FNAME and the LNAME field that are passed in from the trigger buffer.

- ▶ The program checks the application's commit lock level. If it runs under commitment control, the file is opened with commitment control. Otherwise, the file is opened without commitment control.

---

### Example 11-39 Checking the application's commit lock level

---

```
if(strcmp(hstruct->commit_lock_level,"0") == 0)
{
    if (( endaPtr = _Ropen( "R7399_API/ENCDATA",
                          "rr+, riofb=N")) == NULL)
    {
        printf (">>_Ropen ENCDATA \n");
        printf ("    Open of ENCDATA file failed errno %d.\n", errno );
        exit(1);
    }
}
else
{
    if (( endaPtr = _Ropen( "R7399_API/ENCDATA",
                          "rr+,commit=Y")) == NULL)
    {
        printf (">>_Ropen ENCDATA for commit YES \n");
        printf ("    Open of ENCDATA file failed errno %d.\n", errno );
        exit(1);
    }
}
```

---

- ▶ Constructs the database key field from FNAME and LNAME and reads the record from ENCDATA.

---

### Example 11-40 Construction the database key field

---

```
memset (&endain, ' ', sizeof(endain));
memset (&endakey, ' ', sizeof(endakey));

memcpy( endakey.FNAME, &oldbuf, sizeof(oldbuf.FNAME) +
        sizeof(oldbuf.LNAME));

dbendaPtr = _Rreadk( endaPtr, &endain, sizeof(endain), __KEY_EQ,
                   &endakey, sizeof(endakey));

if ( endaPtr->riofb.num_bytes == 0 )
{
    printf (">>Rreadk ENCDATA \n");
    printf ("    No data %.15s in ENCDATA\n", endaout.FNAME);
    _Rclose(endaPtr);
    exit(1);
}
```

---

- ▶ If the record exists in the ENCDATA file and event is DELETE, the record is deleted from the ENCDATA file and returns control to the database manager (Example 11-41).

*Example 11-41 Delete record the ENCDATA file*

---

```

if ((strcmp(hstruct ->trigger_event,"2",1)== 0)) /* delete event */
{
    _Rdelete(endaPtr);          /* delete record from ENCDATA */
    _Rclose(endaPtr);          /* close ENCDATA file */
    return;                    /* exit */
}

```

---

- ▶ If an UPDATE event occurs, the trigger program tests whether the UPDATE statement wants to change FNAME or LNAME. This two fields are unchangeable (Example 11-42).

*Example 11-42 Test for update FBAME and LNAME fields*

---

```

if ( memcmp(&oldbuf,&newbuf, sizeof(newbuf.FNAME) +
           sizeof(newbuf.LNAME) ) != 0 )
{
    printf (">>FNAME and LNAME is changing \n");
    exit(1);
}

```

---

- ▶ The program constructs a new record and recalculates the hash, converts this hash to Base64 form, and finally updates the record in the ENCDATA file (Example 11-43).

*Example 11-43 Construction of new record and hash calculation*

---

```

Length_of_input_data = sizeof(endain) -
                       sizeof(endain.HASH64);

Cryptographic_service_provider = Qc3_Any_CSP;
memset(&Algorithm_description, 0, sizeof(Algorithm_description));
Algorithm_description[3] = Qc3_SHA1;

memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = sizeof(errCode);

Qc3CalculateHash ( (char*)&endain.FNAME,
                  &Length_of_input_data,
                  "DATA0100",
                  (char*)&Algorithm_description,
                  "ALGD0500",
                  &Cryptographic_service_provider,
                  NULL,
                  Hash,
                  &errCode );

if ( errCode.Bytes_Available != 0 )
{
    printf (">>Qc3CalculateHash \n");
    printf ("  errCode.Exception_Id = %.7s\n", errCode.Exception_Id);
    exit(1);
}

len = to_Base64( Hash, sizeof(Hash), endain.HASH64, len );

```

```

printf ("Base64(SHA1(Hash)) = %.28s\n",endain.HASH64 );

if ((_Rupdate( endaPtr, &endain,
              sizeof(endain)))->num_bytes < sizeof(endain))
{
    printf("Update record from ENCDATA failed errno %d.\n" , errno );
    _Rclose(endaPtr);
    exit(1);
}
_Rclose(endaPtr);
return;

```

---

## Execution example of trigger program

This section provides an execution example of our user-written trigger program. This might sound redundant, and it is in a way, but it gives you a hands-on look and feel of the application along with actual messages you might see. Consider this as a summary of an application description with tagged along messages.

The source code for our trigger program is stored in the library R7399\_API in the source physical file QCSRC\_SQL.

Before we start this section, we assume that the library R7399\_API is restored, and that the information from the previous section has been inserted into file ENCDATA.

1. Run Query to see the record in the ENCDATA file:

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/ENCDATA)
```

FNAME	LNAME	ADRES	CITY
000001 Tom	Bliss	1232 Maple St.	Oronoko
000002 Adela	Roubicek	Rumunska 88	Praha
000003 Pavel	Kalhous	Na ruzku 8	Chrudim
000004 Robin	Tatam	Country Rd 59	Chariton

Figure 11-59 List of customers in ENCDATA file

2. Create the physical file EMPDATA:

```

CRTPF FILE(R7399_API/EMPDATA) SRCFILE(R7399_API/QDDSSRC)
      SRCMBR(EMPDATA) TEXT('Employee database file')

```

3. Insert into this database file EMPDATA the same information that we put into the database file ENCDATA, without the secret data (Figure 11-59).

To insert this information, we start SQL:

```
STRSQL
```

Use SQL statements:

```

INSERT INTO R7399_API/EMPDATA
  values('Tom',
        'Bliss',
        '1232 Maple St.',
        'Oronoko',
        'USA',
        'MN',
        '16100',
        '11111')

```

```

INSERT INTO R7399_API/EMPDATA
  values('Adela',
        'Roubicek',
        'Rumunska 88',
        'Praha',
        'Czech Republic',
        'CZ',
        '11022',
        '26588')

```

```

INSERT INTO R7399_API/EMPDATA
  values('Robin',
        'Tatam',
        'Country Rd 59',
        'Chariton',
        'USA',
        'IO',
        '12682',
        '82345')

```

```

INSERT INTO R7399_API/EMPDATA
  values('Pavel',
        'Kalhous',
        'Na ruzku 8',
        'Chrudim',
        'Czech Republic',
        'CZ',
        '52692',
        '12698')

```

4. Run Query to see the list of records put in the database file EMPDATA:

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/EMPDATA)
```

	FNAME	LNAME	ADRES	CITY
000001	Tom	Bliss	1232 Maple St.	Oronoko
000002	Adela	Roubicek	Rumunska 88	Praha
000003	Pavel	Kalhous	Na ruzku 8	Chrudim
000004	Robin	Tatam	Country Rd 59	Chariton

Figure 11-60 RUNQRY QRY(\*NONE) QRYFILE(R7399\_API/EMPDATA)

5. Run Query to see the list of records put in the database file ENCDATA:

```
RUNQRY QRY(*NONE) QRYFILE(R7399_API/ENCDATA)
```

	FNAME	LNAME	ADRES	CITY
000001	Tom	Bliss	1232 Maple St.	Oronoko
000002	Adela	Roubicek	Rumunska 88	Praha
000003	Robin	Tatam	Country Rd 59	Chariton
000004	Pavel	Kalhous	Na ruzku 8	Chrudim

Figure 11-61 RUNQRY QRY(\*NONE) QRYFILE(R7399\_API/ENCDATA)

6. Define an external trigger.

a. Create C program UPD\_TRG\_DT:

```
CRTCMOD MODULE(R7399_API/UPD_TRG_DT)
        SRCFILE(R7399_API/QCSRC_SQL)

CRTPGM PGM(R7399_API/UPD_TRG_DT)
        MODULE(R7399_API/UPD_TRG_DT
              R7399_API/BASE64_COD )
```

b. Add a trigger to a table:

```
ADDPFTRG FILE(R7399_API/EMPDATA)
        TRGTIME(*AFTER) TRGEVENT(*UPDATE)
        PGM(R7399_API/UPD_TRG_DT)
        TRG(UPD_TRG_DT_UPDATE)
        TRGLIB(R7399_API)
        MLTTHDACN(*RUN) TRGUPDCND(*CHANGE)

ADDPFTRG FILE(R7399_API/EMPDATA)
        TRGTIME(*BEFORE) TRGEVENT(*DELETE)
        PGM(R7399_API/UPD_TRG_DT)
        TRG(UPD_TRG_DT_DELETE)
        TRGLIB(R7399_API) MLTTHDACN(*RUN)
```

If you receive the message Trigger operation not successful and if the trigger definition already exists, you have to use the CL command to remove this physical file trigger:

```
RMVPFTRG FILE(R7399_API/EMPDATA) TRG(UPD_TRG_DT_UPDATE) TRGLIB(R7399_API)
RMVPFTRG FILE(R7399_API/EMPDATA) TRG(UPD_TRG_DT_DELETE) TRGLIB(R7399_API)
```

7. Display the hash for record Adela Roubicek:

```
STRSQL

SELECT      FNAME, LNAME,
            HASH_DSP ( FNAME || LNAME || ADRES || CITY ||
                      COUNTRY || STATE || ZIP || PHONE ||
                      IV64 || ENC64 || KEYLAB || QKSFN ||
                      TIME ,
                      char_length( FNAME || LNAME || ADRES || CITY
                                   ||
                                   COUNTRY || STATE || ZIP || PHONE
                                   ||
                                   IV64 || ENC64 || KEYLAB || QKSFN
                                   ||
                                   TIME ),
                      HASH64 ) as Result
            from ENCDATA
            WHERE FNAME = 'Adela'
```

FNAME	LNAME	RESULT
Adela	Roubicek	TN0QG6Vxwt20p5Fe19jc/jNtah0=

8. Update record Adela Roubickova in the EMPDATA file:

```
STRSQL

UPDATE EMPDATA
```



```

SET CITY = 'Kolodeje 29', ADRES = 'Sezemice'
WHERE FNAME = 'Adela'

```

**Base64(SHA1(Hash)) = ZhnUm51WLHFp2pIOYU+EFbJKwNA=**

**Press ENTER to end terminal session.**

1 rows updated in EMPDATA in R7399\_API.

9. Delete record Pavel Kalhous in the EMPDATA file:

STRSQL

```

DELETE FROM EMPDATA
WHERE FNAME = 'Pavel' AND LNAME = 'Kalhous'

```

**1 rows deleted from EMPDATA in R7399\_API.**

10. Run Query to see a list of records put in the database file EMPDATA again:

RUNQRY QRY(\*NONE) QRYFILE(R7399\_API/EMPDATA)

000001	Tom	Bliss	1232 Maple St.	Oronoko
000002	Adela	Roubicek	Sezemice	Kolodeje 29
000004	Robin	Tatam	Country Rd 59	Chariton

Figure 11-62 RUNQRY QRY(\*NONE) QRYFILE(R7399\_API/EMPDATA)

11. Run Query to see a list of records put in the database file ENCDATA again:

RUNQRY QRY(\*NONE) QRYFILE(R7399\_API/ENCDATA)

	FNAME	LNAME	ADRES	CITY
000001	Tom	Bliss	1232 Maple St.	Oronoko
000002	Adela	Roubicek	Sezemice	Kolodeje 29
000003	Robin	Tatam	Country Rd 59	Chariton

Figure 11-63 RUNQRY QRY(\*NONE) QRYFILE(R7399\_API/ENCDATA)

12. Verify the hash for record Adela Roubicek in the ENCDATA file:

STRSQL

```

SELECT FNAME, LNAME,
       HASH_DATA( FNAME || LNAME || ADRES || CITY ||
                  COUNTRY || STATE || ZIP || PHONE ||
                  IV64 || ENC64 || KEYLAB || QKSFN ||
                  TIME ,
                  char_length( FNAME || LNAME || ADRES || CITY
                              ||
                              COUNTRY || STATE || ZIP || PHONE
                              ||
                              IV64 || ENC64 || KEYLAB || QKSFN
                              ||
                              TIME ),
                  HASH64 ) as Result
from ENCDATA
WHERE FNAME = 'Adela'

```

FNAME	LNAME	RESULT
Adela	Roubicek	YES

13. Display a new hash for the record Adela Roubicek in the ENCDATA file:

STRSQL

```

SELECT FNAME, LNAME,
       HASH_DSP ( FNAME || LNAME || ADRES || CITY ||
                 COUNTRY || STATE || ZIP || PHONE ||
                 IV64 || ENC64 || KEYLAB || QKSFN ||
                 TIME ,
                 char_length( FNAME || LNAME || ADRES || CITY
                              ||
                              COUNTRY || STATE || ZIP || PHONE
                              ||
                              IV64 || ENC64 || KEYLAB || QKSFN
                              ||
                              TIME ),
                 HASH64 ) as Result
from ENCDATA
WHERE FNAME = 'Adela'

```

FNAME	LNAME	RESULT
Adela	Roubicek	ZhnUm51WLHFp2pIOYU+EFbJKwNA=



## HW-based method

In this chapter, we provide scenarios to demonstrate a way of using the Cryptographic Coprocessor 4764/4758 hardware with an i5/OS server. The Common Cryptographic Architecture (CCA) API set is used to interface with the Cryptographic Coprocessor 4764/4758 hardware. We use the cryptocard and CCA APIs to encrypt and decrypt data.

## 12.1 Scenario overview

We have two scenarios: scenario A shows the encryption/decryption of data between two systems and scenario B shows the encryption/decryption of data on the same system.

### 12.1.1 Scenario A: exchanging secret data between two systems

On the source system we have a file that includes sensitive data. We want to move this data to the target system over the nonsecure channel or save/restore on media, such as DVD or tape. The basic structure of the scenario is outlined in Figure 12-1.

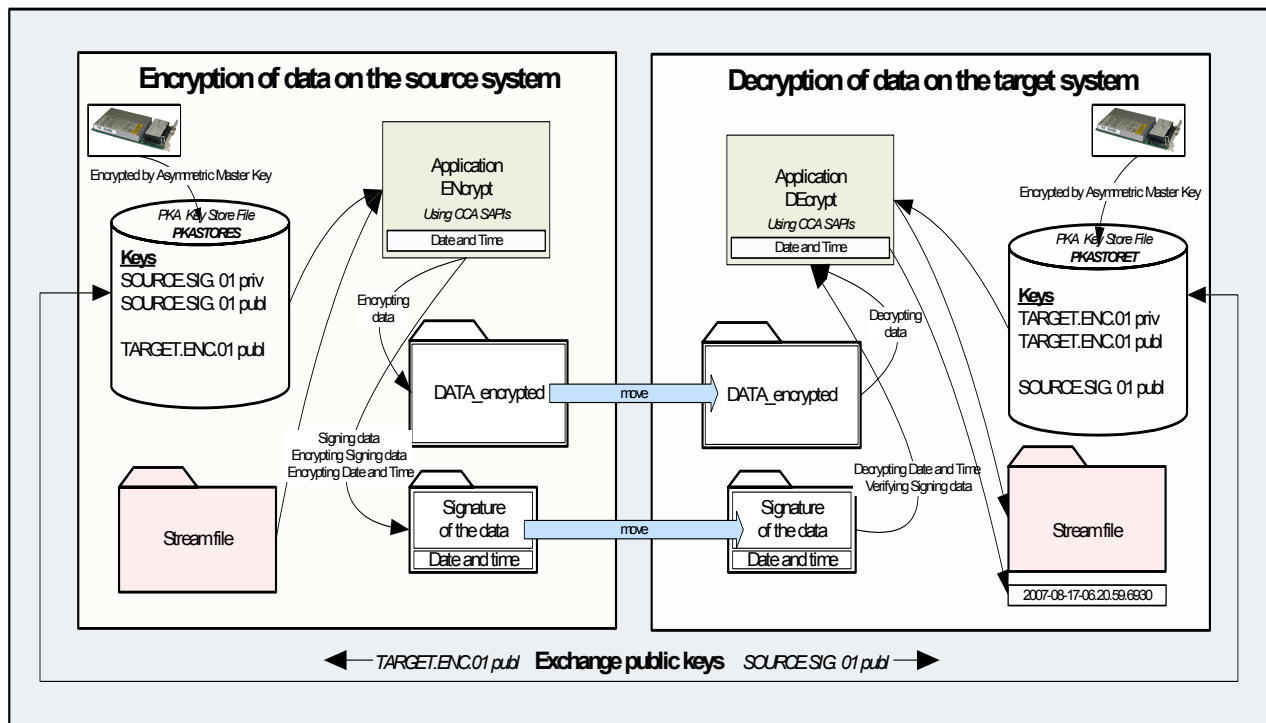


Figure 12-1 Encryption/decryption of data on the two systems: from the source to the target system

On the source system we use symmetric encryption to encrypt a sensitive stream file. To protect data from undetected changes we generate a proof of identity value called a digital signature. We use symmetric encryption to encrypt this digital signature and current date and time. Encrypted data and encrypted digital signature are saved into different stream files. We send/save these two files from the source system to the target system where we receive or restore them.

On the target system we decrypt both files—a stream file with the data and a stream file with a digital signature, the date, and the time. We start a signature process on the source system and a verifying process on the target system to ensure that the data has not been changed during the transfer.

For an encryption/decryption of data, a digital signature, and date and time, we need to generate secret symmetric DES keys (double length DATA key). For the distribution of these generated symmetric DES keys, we generate one pair of asymmetric keys (a private key and a public key) for encryption on the target system in the keystore file. As shown in the diagram, the keystore file is named PKASTORET (PKA Key Store File) and they keys are named TARGET.ENC.01.

For the digital signature/verification process we generate one pair of asymmetric keys for the signing, SOURCE.SIG.01, on the source system in the keystore file PKASTORES.

Finally, we need to exchange the public keys of the asymmetric pairs between the target and the source system. The public key for signing, SOURCE.SIG.01, from the keystore file PKASTORES on the source system is sent to the keystore file PKASTORET on the target system. The public key for encryption TARGET.ENC.01 from the keystore file PKASTORET on the target system is sent to the keystore file PKASTORES on the source system.

In some cases we want to send sensitive data from the opposite side, from the target system to the source system. The basic structure of the scenario is outlined in Figure 12-2. This scenario is logically the same as the previous four paragraphs except that target and source are switched.

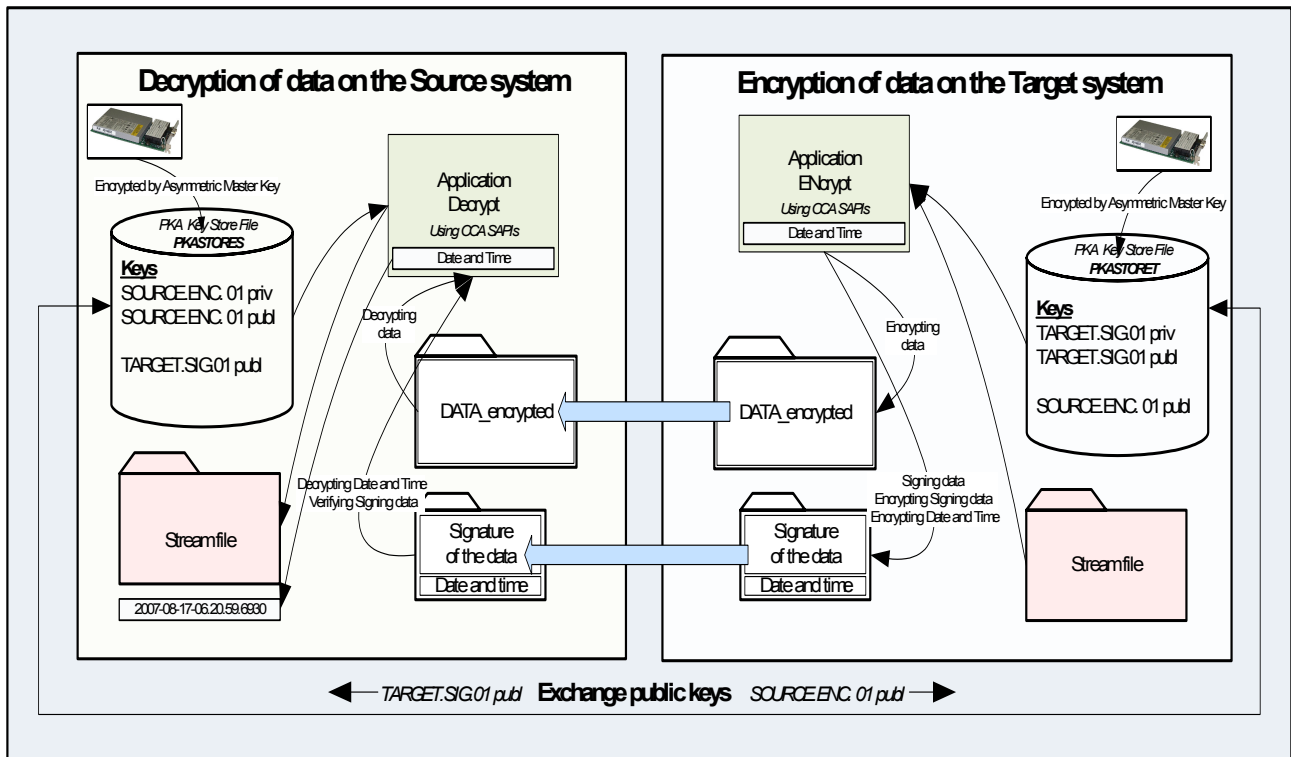


Figure 12-2 Encryption/decryption data on a different system (from the target to the source system)

This is conceptually the same scenario as the case described in Figure 12-1 on page 230 except that target and source are switched.

## 12.1.2 Scenario B: encryption/decryption of data on the same system

We use this scenario if we want to encrypt/decrypt data on the same system or if we have only one system with Cryptographic Coprocessor 4758/4764 as an encryption/decryption engine (accelerator). The scenario scheme is outlined in Figure 12-3.

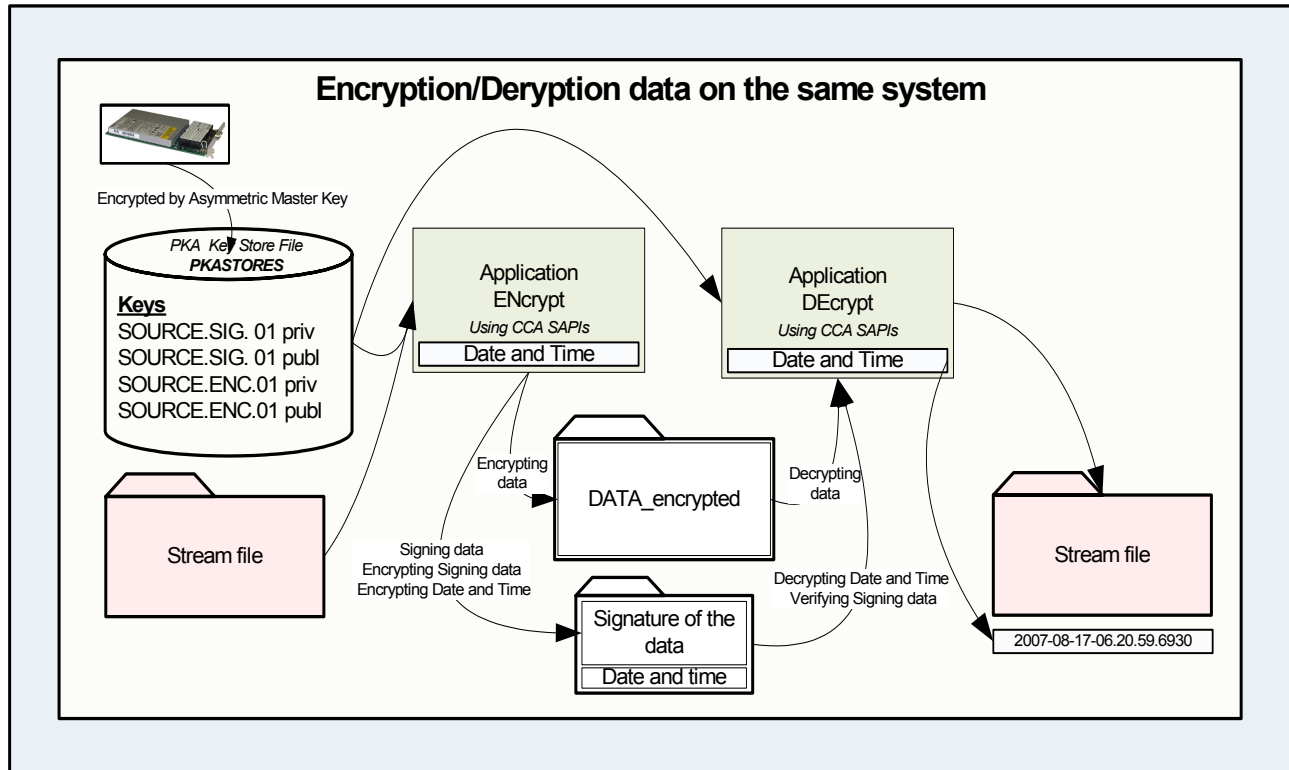


Figure 12-3 Encryption/decryption data on the same system

On the system we use symmetric encryption to encrypt sensitive stream file. To protect data from undetected changes we generate a proof of identity value called a digital signature. We use symmetric encryption to encrypt the digital signature with the date and the time when the file was encrypted (current date and time). Encrypted data and the encrypted digital signature with the date and the time are saved in different stream files.

On the same system we decrypt both files, a stream file with the encrypted data and a stream file with an encrypted digital signature and the date and the time. And, finally, we start verifying and signature process to ensure that the data has not been changed.

For an encryption/decryption data/digital signature we need the date and the time to generate symmetric DES key. For the protection of the generated secret symmetric DES keys we generate one pair of asymmetric keys (public, private), TARGET.ENC.01, for an encryption process in the keystore file PKASTORES. For the digital signature/verification process we generate one pair of asymmetric keys (public, private), SOURCE.SIG.01, in the keystore file PKASTORES.

## 12.2 Prerequisites and assumptions

It is important that we ensure that our source and target systems meet the requirements necessary for the Cryptographic Coprocessor prior to installing it. These requirements include hardware and software prerequisites.

We need to ensure the secure access of our source and target server resources prior to installing a Cryptographic Coprocessor.

Let us familiarize ourselves with the object authorities that are required for the security APIs (SAPI) used in our scenario:

- ▶ Cryptographic\_Resource\_Allocate (CSUACRA)
- ▶ Cryptographic\_Resource\_Deallocate (CSUACRD)
- ▶ Logon\_Control (CSUALCT)
- ▶ Keystore\_Initialize (CSNBKSI)
- ▶ Key\_Storage\_Designate (CSUAKSD)
- ▶ PKA\_Key\_Record\_Create (CSNDKRC)
- ▶ PKA\_Key\_Generate (CSNDPKG)
- ▶ PKA\_Public\_Key\_Extract (CSNDPKX)
- ▶ PKA\_Key\_Record\_Create (CSNDKRC)
- ▶ Digital\_Signature\_Generate (CSNDDSG)
- ▶ One\_Way\_Hash (CSNDDSG)
- ▶ PKA\_Key\_Record\_Read (CSNDKRR)
- ▶ PKA\_Symmetric\_Key\_Generate (CSNDSYG)
- ▶ Encipher (CSNBENC)
- ▶ PKA\_Symmetric\_Key\_Import (CSNDSYI)
- ▶ Digital\_Signature\_Verify (CSNDDSV)
- ▶ Decipher (CSNBDEC)

**Note:** You can find a description of these APIs (SAPI) in the manual *IBM PCI Cryptographic Coprocessor, CCA Basic Services Reference and Guide, Release 2.52, IBM iSeries PCICC Feature*, which can be found at the following link:

[http://www-306.ibm.com/security/cryptocards/pdfs/IBM\\_4758\\_Basic\\_Services\\_Release\\_2\\_52.pdf](http://www-306.ibm.com/security/cryptocards/pdfs/IBM_4758_Basic_Services_Release_2_52.pdf)

Appendix A of the manual describes the return codes and the reason codes that a verb uses to report the results of processing.

To prepare the scenario we must perform the following steps:

1. The source and the target system have a 4764/4758 Cryptographic Coprocessor installed and configured properly.
2. Install the 4764/4758 Cryptocard hardware and follow the instructions of the Cryptographic Coprocessor Configuration Wizard.
3. Create roles and profiles in the 4758/4764 Cryptographic Coprocessor to allow us to access APIs (SAPI) used in our scenario.
4. We assume that the Cryptographic Coprocessor card with the profile is already identified either by defaulting to the CRP01 device or by being explicitly named using the call program CRPALLOC. Also, this device must be varied on, and we must be authorized to use this device description.

**Note:** Cryptographic Coprocessors on systems running the i5/OS operating system use role-based access control. In a role-based system, you define a set of roles that correspond to the classes of coprocessor users. You can enroll each user by defining an associated user profile to map the user to one of the available roles.

The capabilities of a role are dependent on the access control points or cryptographic hardware commands that are enabled for that role. You can then use your Cryptographic Coprocessor to create profiles that are based on the role that you choose.

As you design your application, consider the commands that you must enable or restrict in the access-control system and the implications to your security policy.

5. Every Cryptographic Coprocessor must have a role called the default role. Any user that has not logged on to the Cryptographic Coprocessor will operate with the capabilities defined in the default role. Users who only need the capabilities defined in the default role do not need a profile. In most applications, the majority of the users will operate under the default role, and will not have user profiles. Typically, only security officers and other special users need profiles.

In our scenarios, we need an API that uses an access control point that is not enabled in the default role. That is why we log on with a profile that uses a role having the access control point for particular APIs.

**Note:** There is a performance impact when we log on to the Cryptocard. Key management functions should be enabled for a role other than the default role. The default role should be enabled for encrypt/decrypt or sign/verify. In this way you can log off the card and then still do the encrypt/decrypt or sign/verify without the performance impact.

## 12.3 Scenario environment setup

To build the scenario environment:

1. Go to this Redbooks publication's Web site for download. The following are in the downloaded package:

- R7399\_CCA.savf
- text01.txt
- text02.txt

2. Restore library R7399\_CCA.
3. Compile the CL program CRTPGM:

```
CRTCLPGM PGM(R7399_CCA/CRTPGM)
          SRCFILE(R7399_CCA/QCLSRC)
          SRCMBR(CRTPGM)
          TEXT(*SRCMBRTXT) REPLACE(*YES) TGTRLS(*PRV)
```

4. To create all programs for our scenarios, run the program CRTPGM:

```
CALL PGM(R7399_CCA/CRTPGM)
```

**Note:** When we restore library R7399\_CCA, all programs needed for our scenarios are restored in this library.



5. Create a directory in the source system where <user profile> is MilanK:

```
MKDIR DIR('/home/<user profile>')
MKDIR DIR('/home/<user profile>/Files')
```

Create a directory in the target system where <user profile> is CZ50257:

```
MKDIR DIR('/home/<user profile>')
MKDIR DIR('/home/<user profile>/Files')
```

6. Restore files that we want to encrypt to the directory /home/<user profile>/Files. In our scenarios we restore the files text01.txt and text02.txt.

## 12.4 Exchanging secret data between two systems (scenario A)

In this section we provide a step-by-step guide to encrypt and decrypt secret data between two systems. Refer to the scenario outlined in Figure 12-1 on page 230 and Figure 12-2 on page 231.

### Process summary

This section highlights the tasks to be performed in this scenario:

- ▶ Source system setup and keys generation
- ▶ Transferring files with public keys from source to target system
- ▶ Target system setup and keys generation
- ▶ Transferring files with public keys from target to source system
- ▶ Adding public keys from target onto source system's keystore file
- ▶ Encrypting data
- ▶ Transferring encrypted files to target system
- ▶ Logging off and deallocating on source system
- ▶ Decrypting data
- ▶ Logging off and deallocating on target system

### 12.4.1 Two systems scenario: step-by-step guide

This section provides detailed information for each step to be taken on the source system, summarized in the previous section.

#### Source system setup and keys generation

To set this up:

1. Add the library R7399\_CCA to the user portion of the library list for the our job:

```
ADDLIBLE LIB(R7399_CCA)
```

2. Allocate the Cryptographic Coprocessor device description to our job:

```
CALL PGM(CRPALLOC) PARM(CRP02)
```

This program uses the Cryptographic\_Resource\_Allocate (CSUACRA) API verb to explicitly allocate a cryptographic device to our job so that the system can determine how to route all subsequent cryptographic requests.

If we use any of the CCA API verbs without first explicitly using the Cryptographic\_Resource\_Allocate (CSUACRA) API verb, the system will attempt to allocate the default cryptographic device. The default device is the cryptographic device named CRP01. It must be created by either using the Basic Configuration wizard or the Create Device Crypto (CRTDEVCRP) CL command.

We only need to use CSUACRA when we wish to use a device other than the default cryptographic device. A device allocated to a job, either explicitly or implicitly, remains allocated until either the job ends or the device is deallocated using the Cryptographic\_Resource\_Deallocate (CSUACRD) API verb.

**Note:** The program CRPALLOC is included also in manual *System i Networking Cryptographic hardware Version 5 Release 4*:

<http://publib.boulder.ibm.com/infocenter/iseres/v5r4/topic/rzajc/rzajc.pdf>

3. Log on into the Cryptographic Coprocessor.

We can log on by using the program LOGON that uses the Logon\_Control (CSUALCT) API verb. To start this program we enter the command:

```
CALL PGM(LOGON) PARM('user ID' 'passphrase')
```

Where 'user ID' is a string that identifies the user to the system and 'passphrase' is data used in the authentication process.

We need to log on only if we wish to use APIs that use access control points that are not enabled in the default role. Log on with a profile that uses a role that has the access control points we want to use enabled.

**Note:** The program LOGON is included also in manual *System i Networking Cryptographic Hardware Version 5 Release 4*:

<http://publib.boulder.ibm.com/infocenter/iseres/v5r4/topic/rzajc/rzajc.pdf>

4. Generate two pairs asymmetric keys pairs and store both pairs to the keystore file. Extract the public keys from this keystore file, store them to the stream files, and send the files to the target system.

These steps appear in Figure 12-4. We created the program PKAKEYGEN.

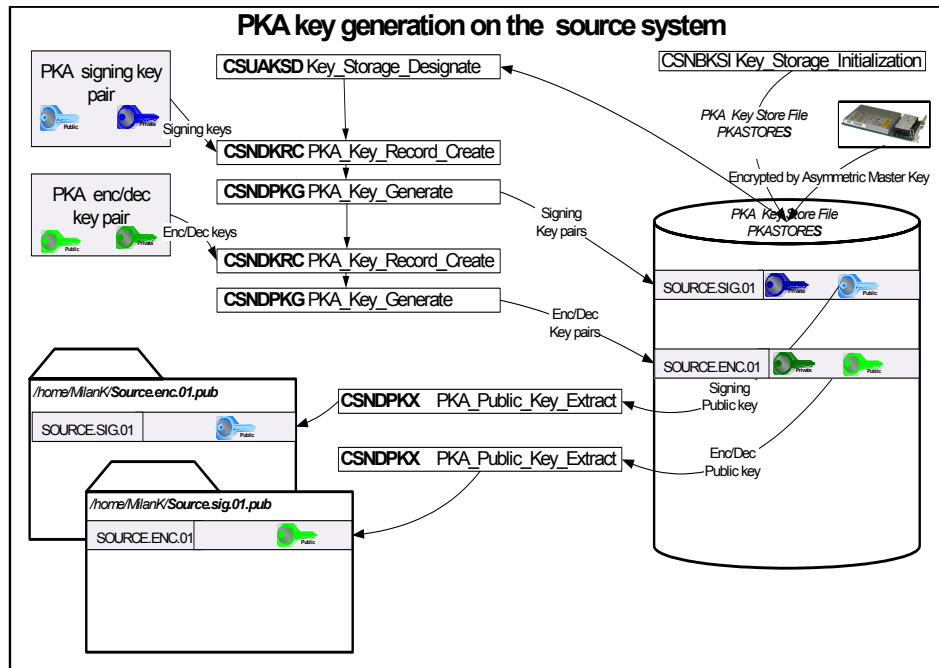


Figure 12-4 Generation of the asymmetrical key pairs on the source system

Run the following command:

```
CALL PGM(PKAKEYGEN)
     PARM(' *YES '
          'SOURCE.ENC.01 '
          'SOURCE.SIG.01 '
          'PKASTORES R7399_CCA '
          ' *YES '
          '/home/MilanK/Source.enc.01.pub '
          '/home/MilanK/Source.sig.01.pub')
```

The parameters and their values used in this program execution are:

- \*YES: A keystore file, R7399\_CCA/PKASTORES, is to be initialized.
- SOURCE.ENC.01: key label of asymmetric key pair for encryption.
- SOURCE.SIG.01: key label of asymmetric key pair for signing.
- PKASTORES R7399\_CCA: a fully qualified file name of the keystore file.
- \*YES: This is to extract public keys and store them to the stream files.
- /home/MilanK/Source.enc.01.pub: A stream file where an extracted public key for encryption is to be stored.
- /home/MilanK/Source.sig.01.pub: A stream file where an extracted public key for signature verification is to be stored.

The program PKAKEYGEN creates two pairs of asymmetric keys as two records in a keystore file, PKASTORES (Example 12-1).

**Note:** Enter this command to verify that two key pairs are created.:

```
QSYS/DSPPFM FILE(R7399_CCA/PKASTORES) MBR(QAC6PKEY)
```

*Example 12-1 DSPPFM of keystore file PKASTORES.*

---

```

Display Physical File Member
File . . . . . : PKASTORES           Library . . . . . : R7399_CCA
Member . . . . . : QAC6PKEY           Record . . . . . : 1
Control . . . . .           Column . . . . . : 1
Find . . . . .
*...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
SOURCE ENC 01
SOURCE SIG 01
***** END OF DATA *****

```

---

### **Description of PKAKEYGEN program**

In this section we provide a description of the PKAKEYGEN program.

► Initialization of a keystore file

If the first parameter is \*YES or keystore file R7399\_CCA/PKASTORES does not exist, a keystore file R7399\_CCA/PKASTORES is initialized.

If the keystore file already exists and this first parameter is \*NO, generated key pairs will be stored in the existing keystore file.

If the keystore file R7399\_CCA/PKASTORES already exists and the first parameter is \*YES, a keystore file R7399\_CCA/PKASTORES is initialized again and all keys in this keystore file are deleted.

To create a keystore file, the Key\_Storage\_Initialization verb CSNBKSI is used. The Key\_Storage\_Initialization verb initializes a key-storage file using the current asymmetric master-key.

► Designation of the key-storage file

Before we can perform any operation using the keystore file R7399\_CCA/PKASTORES or keys stored in this keystore file, we must name the keystore file by the SAPI the Key\_Storage\_Designate CSUAKSD.

► Generation of one asymmetric key pair for encrypting and storing this key pair to the keystore file

We generate one asymmetrical key pair for encryption in three steps:

- a. The first step we use is the PKA\_Key\_Record\_Create CSNDKRC verb. This service adds a key record with a null key-token to the keystore file (PKA key-storage). The new key-record is identified by the key label SOURCE.ENC.01 and is specified as the second parameter in our program.
- b. In the second step we build a key token, needed to generate the PKA key for encryption. The RSA Key-Token Header Token identifier X'1E' specifies an external token. The optional private-key is either in cleartext or enciphered by a transport key-encrypting key. Section identifier X'02' specifies RSA private key, 1024-Bit modulus-exponent form. The key usage flag bits X'80' indicate that signature generation and key decryption are allowed. Section identifier X'04' specifies the RSA

public key with the Public-key exponent field length in bytes 3, Public-key modulus length in 1024 bits, and Public-key exponent 65537.

- c. In the third step we use the PKA\_Key\_Generate verb (CSNDPKG) to generate a public-private encryption key-pair for use with the RSA algorithm. The generated private-key is returned in the form enciphered by the CCA asymmetric master-key.

The generated asymmetric key pair for encryption with the key label identification SOURCE.ENC.01 is stored to the keystore file PKASTORES.

- ▶ Extracting the public encrypting key from the keystore file and storing this public key to the stream file

We extract the public key with key label identification SOURCE.ENC.01 from the keystore file PKASTORES, and we store this key to the stream file Source.enc.01.pub. To extract this public key we use PKA\_Public\_Key\_Extract verb CSNDPKX.

The first 64 bytes of the stream file structure contain the label of the key SOURCE.ENC.01. The next parameter of stream file structure is an integer variable that contains the number of bytes of data in the key\_token variable value. The last parameter of the stream file structure is the key token that will be written to the keystore file (PKA key-storage) on the target system (SAPI PKA\_Key\_Record\_Create verbs CSNDKRC in program STRPUBKEY). The structure of this stream file is shown in Example 12-2.

*Example 12-2 Stream structure of the file containing key token (public key)*

---

```
typedef _Packed struct target_entryStr {
    char label_key[64];
    long target_key_token_length;
    char target_key_token[2500];
} target_entryStr;
```

---

- ▶ Generation of one asymmetric signing key pair and storing this key pair to the keystore file in the same way as an asymmetric key pair for encryption

We generate one pair asymmetric signing key pair in three steps:

- a. The first step we use is the PKA\_Key\_Record\_Create CSNDKRC verb. This service adds a key record with a null key-token to keystore file (PKA key-storage). The new key-record is identified by the key label SOURCE.SIG.01 and is specified as the third parameter in our program.
- b. In the second step we build a key token, needed to generate PKA key encryption.
- c. In the third step we use the PKA\_Key\_Generate verb (CSNDPKG) to generate a public-private signing key-pair for use with the RSA algorithm. The generated private-key is returned in the form enciphered by the CCA asymmetric master-key.

The generated asymmetric key pair for signing with the key label identification SOURCE.SIG.01 is stored to the keystore file PKASTORES.

- ▶ Extracting the public encrypting key from the keystore file and storing public key to the stream file

We extract the public signing key with key label identification SOURCE.SIG.01 from the keystore file PKASTORES and we store this public key to the stream file Source.sig.01.pub. To extract this public key we use the PKA\_Public\_Key\_Extract verb.

The first 64 bytes of the stream file structure contain the label of the key SOURCE.SIG.01. The next parameter of the stream file structure is an integer variable containing the number of bytes of data in the key\_token variable value. The last parameter of the stream file structure is a pointer to a string variable containing the key token that will be written to PKA key-storage on the target system. The structure of this stream file is shown in Example 12-2.

This is the end of PKAKEYGEN program description.

## Transferring files with public keys from source to target system

Now we transfer two stream files holding public keys to the target system.

For a security reason in our lab's environment, we were not able to establish direct connection between the source and the target system. Therefore, we used a PC to transfer data between the source and the target system (Figure 12-5).

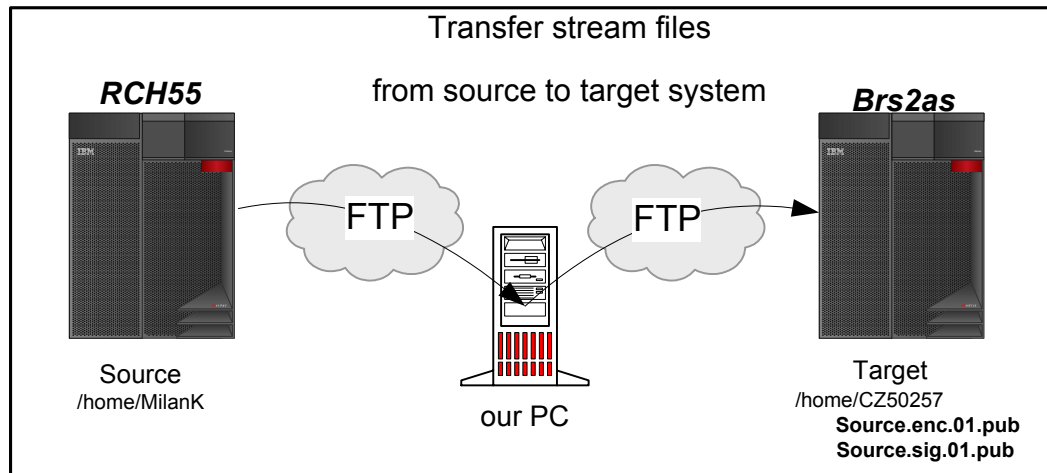


Figure 12-5 Transfer data from the source to the target system

To transfer stream files Source.enc.01.pub and Source.sig.01.pub from the source system RCH55 to our PC and from the PC to the target system Brs2as we used FTP commands (Example 12-3).

Example 12-3 FTP commands to transfer stream files from the source to the target

```
C:\>ftp rch55
Connected to RCH55.
220-QTCP at RCHAS55.RCHLAND.IBM.COM.
220 Connection will close if idle more than 5 minutes.
User (RCH55:(none)): milank
331 Enter password.
Password:
230 MILANK logged on.
ftp> bin
ftp> quote site namefmt 1
250 Now using naming format "1".
ftp> lcd D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC
Local directory now D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC.
ftp> get /home/Milank/Source.enc.01.pub
200 PORT subcommand request successful.
150 Retrieving file /home/Milank/Source.enc.01.pub
226 File transfer completed successfully.
ftp: 2568 bytes received in 0,06Seconds 41,42Kbytes/sec.
ftp> get /home/Milank/Source.sig.01.pub
200 PORT subcommand request successful.
150 Retrieving file /home/Milank/Source.sig.01.pub
226 File transfer completed successfully.
ftp: 2568 bytes received in 0,00Seconds 2568000,00Kbytes/sec.
```

```

ftp>close
ftp> open brs2as
Connected to brs2as.praha.cz.ibm.com.
220-QTCP at BRS2AS.
220 Connection will close if idle more than 5 minutes.
User (brs2as.praha.cz.ibm.com:(none)): cz50257
331 Enter password.
Password:
230 CZ50257 logged on.
ftp> bin
200 Representation type is binary IMAGE.
ftp> quote site namefmt 1
250 Now using naming format "1".
ftp> lcd D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC
Local directory now D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC.
ftp> cd /home/CZ50257
250 "/home/CZ50257" is current directory.
ftp> put Source.enc.01.pub
200 PORT subcommand request successful.
150 Sending file to /home/CZ50257/Source.enc.01.pub
250 File transfer completed successfully.
ftp: 2568 bytes sent in 0,00Seconds 2568000,00Kbytes/sec.
ftp> put Source.sig.01.pub
200 PORT subcommand request successful.
150 Sending file to /home/CZ50257/Source.sig.01.pub
250 File transfer completed successfully.
ftp: 2568 bytes sent in 0,00Seconds 2568000,00Kbytes/sec.
ftp> close
221 QUIT subcommand received.

```

---

## Target system setup and keys generation

Now, we move to the target system for initial setup and keys generation:

1. Add the library R7399\_CCA to the user portion of the library list for the our job:

```
ADDLIBLE LIB(R7399_CCA)
```

2. Allocate a Cryptographic Coprocessor device description to our job.

To Allocate a Cryptographic Coprocessor device description to our job, we need enter the command:

```
CALL PGM(CRPALLOC) PARM(CRP03)
```

This program uses the Cryptographic\_Resource\_Allocate (CSUACRA) API verb to explicitly allocate a cryptographic device to our job so that the system can determine how to route all subsequent cryptographic requests.

If we use any of the CCA API verbs without first explicitly using the Cryptographic\_Resource\_Allocate (CSUACRA) API verb, the system will attempt to allocate the default cryptographic device. The default device is the cryptographic device named CRP01. It must be created by either using the Basic Configuration wizard or the Create Device Crypto (CRTDEVCRP) CL command.

We only need to use CSUACRA when we wish to use a device other than the default cryptographic device. A device allocated to a job, either explicitly or implicitly, remains allocated until either the job ends or the device is deallocated using the Cryptographic\_Resource\_Deallocate (CSUACRD) API verb.

**Note:** The program CRPALLOC is included also in the manual *System i Networking Cryptographic hardware Version 5 Release 4:*

<http://publib.boulder.ibm.com/infocenter/series/v5r4/topic/rzajc/rzajc.pdf>

3. Log on into the Cryptographic Coprocessor.

We can log on by using the program LOGON that uses the Logon\_Control (CSUALCT) API verb. To start this program we enter the command.

```
CALL PGM(LOGON) PARM('user ID' 'passphrase')
```

Where 'user ID' is string that identifies the user to the system and 'passphrase' is data used in the authentication process.

We need to log on only if we wish to use an API that uses access control points that are not enabled in the default role. Log on with a profile that uses a role that has the access control point that we want to use enabled.

**Note:** The program LOGON is included also in the manual *System i Networking Cryptographic hardware Version 5 Release 4:*

<http://publib.boulder.ibm.com/infocenter/series/v5r4/topic/rzajc/rzajc.pdf>

4. Generate two asymmetric key pairs and store both pairs' keys (private,public) to the keystore file and extract the public keys from this keystore file. Store them to the stream files and send them to the source system.

These steps appear are shown in Figure 12-6. For this purpose we write the program PKAKEYGEN. This is the same program that we use on the source system.

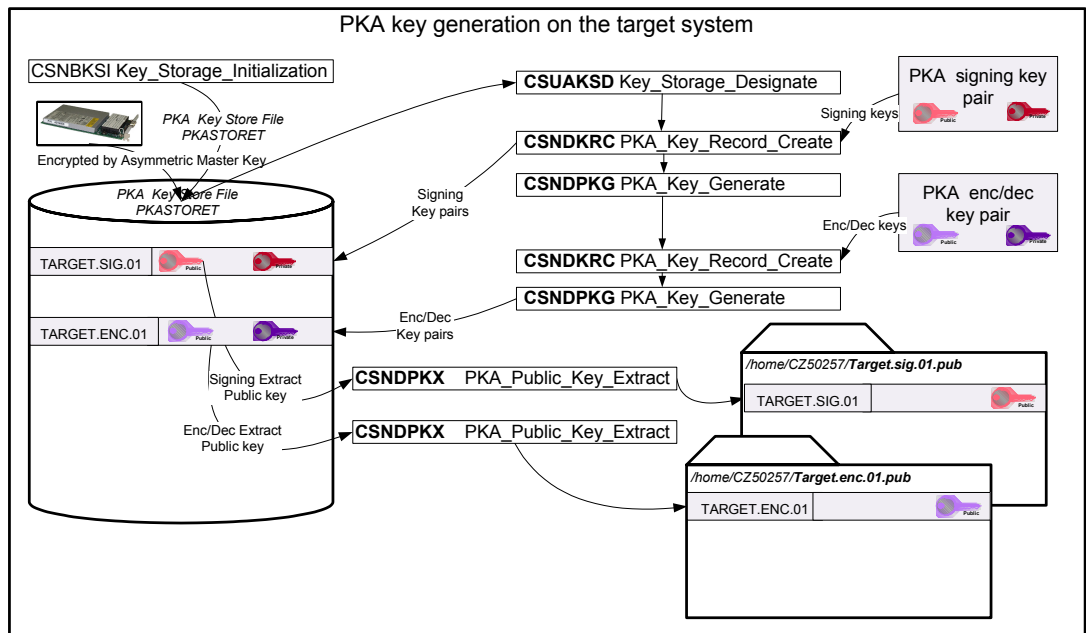


Figure 12-6 Generation of the asymmetrical key pairs on the target system



Run the following command:

```
CALL PGM(PKAKEYGEN)
      PARM('*YES'
           'TARGET.ENC.01'
           'TARGET.SIG.01'
           'PKASTORET R7399_CCA '
           '*YES'
           '/home/CZ50257/Target.enc.01.pub'
           '/home/CZ50257/Target.sig.01.pub')
```

The parameters and their values used in this program execution are:

- \*YES: A keystore file, R7399\_CCA/PKASTORET, is to be initialized.
- TARGET.ENC.01: Key label of asymmetric key pair for encryption.
- TARGET.SIG.01: Key label of asymmetric key pair for signing.
- PKASTORET R7399\_CCA: A fully qualified file name of the keystore file.
- \*YES: This is to extract public keys and store them to the stream files.
- /home/CZ50257/Target.enc.01.pub: A stream file where extracted public key for encryption is to be stored.
- /home/CZ50257/Target.sig.01.pub: A stream file where extracted public key for signing is to be stored.

The program PKAKEYGEN creates in keystore file (PKA key-store file) PKASTORET two records, two key pairs of asymmetric keys (Example 12-4).

*Example 12-4 DSPPFM of keystore file PKASTORET*

---

```
                Display Physical File Member
File . . . . . : PKASTORET          Library . . . . . : R7399_CCA
Member . . . . . : QAC6PKEY         Record . . . . . : 1
Control . . . . . :                  Column . . . . . : 1
Find . . . . . :
*...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
TARGET ENC 01
TARGET SIG 01
                ***** END OF DATA *****
```

---

**Description of PKAKEYGEN program**

The PKAKEYGEN program is as follows:

- Initialization of a keystore file.

If the first parameter is \*YES or keystore file R7399\_CCA/PKASTORET does not exist, keystore file R7399\_CCA/PKASTORET is initialized.

If the keystore file R7399\_CCA/PKASTORET already exists and the first parameter is \*YES, new keystore file R7399\_CCA/PKASTORET is initialized again and all keys into this keystore file are deleted.

If the keystore file already exists and this parameter is \*NO, generated key pairs will be stored in existing keystore file.

To create a keystore file we use the Key\_Storage\_Initialization verb CSNBKSI. The Key\_Storage\_Initialization verb initializes a key-storage file using the current asymmetric master-key.

- ▶ Designate the key-storage file.

For the job that ran our program, before we can perform any operation using the keystore file R7399\_CCA/PKASTORET or keys stored in this keystore file, we must name the keystore file by the SAPI the Key\_Storage\_Designate CSUAKSD.

- ▶ Generate one pair asymmetric key pair for encrypting and store this key pair to the keystore file.

We generate one asymmetric encryption key pair in three steps:

- The first step we use is the PKA\_Key\_Record\_Create CSNDKRC verb. This service adds a key record with a null key-token to PKA key-storage. The new key-record is identified by the key label TARGET.ENC.01 specified as a second parameter of our program.
- In the second step we build a key token, needed to generate PKA key encryption. The RSA Key-Token Header Token identifier X'1E' specifies external token. The optional private-key is either in cleartext or enciphered by a transport key-encrypting key. Section identifier X'02' specifies the RSA private key, 1024-Bit modulus-exponent form. The key usage flag bits X'80' indicates that signature generation and key decryption is allowed. Section identifier X'04' specifies the RSA public key with the Public-key exponent field length in bytes 3, Public-key modulus length in 1024 bits, and Public-key exponent 65537.
- In the third step we use the PKA\_Key\_Generate verb (CSNDPKG) to generate a public-private encryption key-pair for use with the RSA algorithm. The generated private-key is returned in the form enciphered by the CCA asymmetric master-key.

The generated asymmetric encrypting pair key with the key label identification TARGET.ENC.01 is stored to the keystore file PKASTORET.

- ▶ Extract the public encrypting key from the keystore file and store public key to the stream file.

We extract the public encrypting key with key label identification TARGET.ENC.01 from the keystore file PKASTORET and we store this public key to the Stream file Target.enc.01.pub. Structure of this stream file is seen in Example 12-2 on page 239. To extract this public key we use PKA\_Public\_Key\_Extract verb CSNDPKX.

- ▶ Generate one asymmetric key pair for signing and store this key pair to the keystore file in the same way as an asymmetric encryption key pair.

We generate one pair asymmetric signing key pair in three steps.

- The first step we use is PKA\_Key\_Record\_Create CSNDKRC verb. This service adds a key record with a null key-token to PKA key-storage. The new key-record is identified by the key label TARGET.SIG.01 specified as a third parameter of our program.
- In the second step we build a key token, needed to generate PKA key encryption.
- In the third step we use PKA\_Key\_Generate verb (CSNDPKG) to generate a public-private signing key-pair for use with the RSA algorithm. The generated private-key is returned in the form enciphered by the CCA asymmetric master-key.

The generated asymmetric signing pair key with the key label identification TARGET.SIG.01 is stored in the keystore file PKASTORET.

- ▶ Extract the public signing key from the keystore file and store the public key to the stream file.

We extract the public signing key with key label identification TARGET.SIG.01 from the keystore file PKASTORET and we store this public key to the stream file Target.sig.01.pub. The structure of this stream file is shown in Example 12-2 on page 239. To extract this public key we use the PKA\_Public\_Key\_Extract verb.

This is the end of PKAKEYGEN program description.

## Transferring files with public keys from target to source system

Now we transfer two stream files holding public keys to the source system.

For security reasons, in our environment we are not able to establish a direct connection between the target and the source system, so we have to transfer data between the target and the source system over our PC (Figure 12-7).

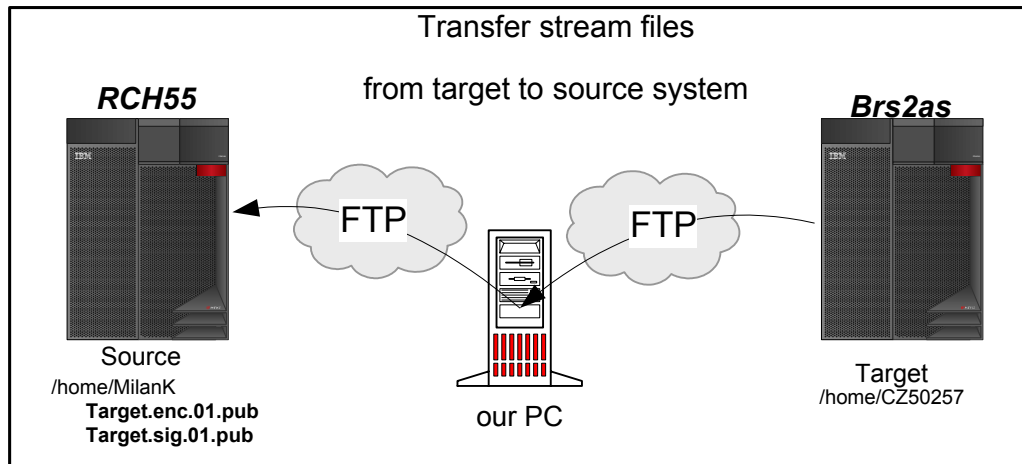


Figure 12-7 Transfer data from the target to the source system

To transfer stream files Target.enc.01.pub and Target.sig.01.pub from the target system Brs2as to our PC and from the PC to the source system RCH55 we use FTP commands (Example 12-5.)

Example 12-5 stream files from the target to the source

```
C:\>ftp Brs2as
Connected to Brs2as.praha.cz.ibm.com.
220-QTCP at BRS2AS.
220 Connection will close if idle more than 5 minutes.
User (Brs2as.praha.cz.ibm.com:(none)): cz50257
331 Enter password.
Password:
230 CZ50257 logged on.
ftp> bin
200 Representation type is binary IMAGE.
ftp> quote site namefmt 1
250 Now using naming format "1".
ftp> lcd D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC
Local directory now D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC.
ftp> get /home/CZ50257/Target.enc.01.pub
200 PORT subcommand request successful.
150 Retrieving file /home/CZ50257/Target.enc.01.pub
250 File transfer completed successfully.
ftp: 2568 bytes received in 0,02Seconds 160,50Kbytes/sec.
ftp> get /home/CZ50257/Target.sig.01.pub
200 PORT subcommand request successful.
150 Retrieving file /home/CZ50257/Target.sig.01.pub
250 File transfer completed successfully.
```

```

ftp: 2568 bytes received in 0,02Seconds 160,50Kbytes/sec.
ftp> close
ftp> open RCH55
Connected to RCH55.
220-QTCP at RCHAS55.RCHLAND.IBM.COM.
220 Connection will close if idle more than 5 minutes.
User (RCH55:(none)): milanK
331 Enter password.
Password:
230 MILANK logged on.
ftp> bin
200 Representation type is binary IMAGE.
ftp> quote site namefmt 1
250 Now using naming format "1".
ftp> lcd D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC
Local directory now D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC.
ftp> cd /home/MilanK
250 "/home/MilanK" is current directory.
ftp> put Target.enc.01.pub
200 PORT subcommand request successful.
150 Sending file to /home/MilanK/Target.enc.01.pub
226 File transfer completed successfully.
ftp: 2568 bytes sent in 0,00Seconds 2568000,00Kbytes/sec.
ftp> put Target.sig.01.pub
200 PORT subcommand request successful.
150 Sending file to /home/MilanK/Target.sig.01.pub
226 File transfer completed successfully.
ftp: 2568 bytes sent in 0,00Seconds 2568000,00Kbytes/sec.
ftp> close
221 QUIT subcommand received.
ftp> quit

```

## Adding public keys from target onto source system's keystore file

Now we add public keys from the transferred stream files from the target system to the keystore file. Transferred files Target.enc.01.pub and Target.sig.01.pub, including the public encryption and signing key, were add in the keystore file PKASTORES (Figure 12-8).

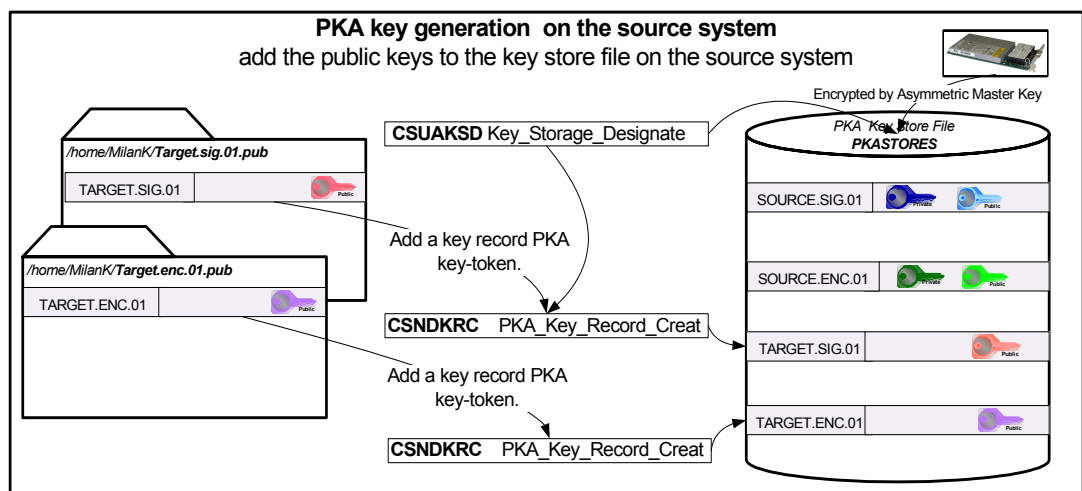


Figure 12-8 Add public keys to the keystore file on the source system

Run the following command:

```
CALL PGM(STRPUBKEY)
      PARM('PKASTORES R7399_CCA '
           '/home/MilanK/Target.enc.01.pub'
           '/home/MilanK/Target.sig.01.pub')
```

The parameters and their values used in this program execution are:

- ▶ **PKASTORES R7399\_CCA**: a fully qualified file name of the keystore file
- ▶ **/home/MilanK/Target.enc.01.pub**: stream file that holds an extracted public key for encrypting
- ▶ **/home/MilanK/Target.sig.01.pub**: stream file that holds an extracted public key for signing

The program STRPUBKEY adds two records—two public keys transferred from the target system (Example 12-6). Now the keystore file PKASTORES contains all needed keys for encrypting the data.

*Example 12-6 QSYS/DSPPFM FILE(R7399\_CCA/PKASTORES) MBR(QAC6PKEY)*

---

```
Display Physical File Member
File . . . . . : PKASTORES           Library . . . . . : R7399_CCA
Member . . . . . : QAC6PKEY           Record . . . . . : 1
Control . . . . . :                   Column . . . . . : 1
Find . . . . . :
*...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
SOURCE ENC      01
SOURCE SIG      01
TARGET ENC     01
TARGET SIG     01
***** END OF DATA *****
```

---

### **Description of STRPUBKEY program**

For the STRPUBKEY program:

- ▶ Designate the key-storage file.  
For the job that ran our program before we can perform any operation using the keystore file or keys stored in this keystore file, we must name the keystore file by the SAPI the Key\_Storage\_Designate CSUAKSD. The keystore file name R7399\_CCA/PKASTORES is specified as the first parameter of the program STRPUBKEY. In the next two steps we store the transferred public keys for encryption/signing in this keystore file.
- ▶ Add the public encryption key to the keystore file name.  
We retrieve information from the stream file Target.enc.01.pub specified as the second parameter of the program STRPUBKEY and use SAPI PKA\_Key\_Record\_Create CSNDKRC to store the public encryption key to the keystore file PKASTORES. The structure of the file Target.enc.01.pub is shown in Example 12-2 on page 239.
- ▶ Add the public signing key to the keystore file name.  
We retrieve information from the stream file Target.sig.01.pub specified as the third parameter of the program STRPUBKEY and use SAPI PKA\_Key\_Record\_Create CSNDKRC to store the public signing key to the keystore file PKASTORES. The structure of the file Target.sig.01.pub is shown in Example 12-2 on page 239.

This is the end of STRPUBKEY program description.

## Encrypting data

Now we encrypt data on the source system. In our scenario we encrypt the stream text file. As a result of our encryption process we have two new files. The first file holds the encrypted text file and it has the suffix <enc>. The second file holds the encrypted digital signature of the text file and the current date and time. It has the suffix <sig>.

For encryption of the text file and digital signing and the current date and the time we use generated random double length DATA key. This DATA key is sent together with the encrypted text file and the digital signature, date, and time. This DATA key is in an encrypt form and it is encrypted by asymmetric public key for encryption. The structures of these files are shown in Figure 12-11 on page 251 and Figure 12-13 on page 253.

Run the following command.

```
CALL PGM(ENCDATA)
      PARM('/home/MilanK/Files/text01.txt'
           '/home/MilanK/Files/text01.sig'
           '/home/MilanK/Files/text01.enc'
           'PKASTORES R7399_CCA '
           'SOURCE.SIG.01'
           'TARGET.ENC.01' )
```

The parameters and their values used in this program execution are:

- ▶ /home/MilanK/Files/text01.txt: text file to be encrypted
- ▶ /home/MilanK/Files/text01.sig: encrypted digital signature of the text file and the current date and time
- ▶ /home/MilanK/Files/text01.enc: encrypted text file
- ▶ PKASTORES R7399\_CCA: a fully qualified file name of the keystore file
- ▶ SOURCE.SIG.01: key label of the private key for signing of the digest of the text file
- ▶ TARGET.ENC.01: key label of the public key for encryption DATA keys

### ***Description of ENCDATA program***

For the ENCDATA program:

- ▶ Designate the key-storage file.

For the job that ran our program, before we can perform any operation using the keystore file or keys stored in this keystore file, we must name the keystore file by the SAPI the Key\_Storage\_Designate CSUAKSD. The keystore file name R7399\_CCA/PKASTORES is specified as the fourth parameter of the program ENCDATA.

- Retrieve the current date and time and generate the digital signature of the data that will be encrypted.

We can protect data from undetected modification by including a proof-of-data-integrity value. This proof-of-data-integrity value is called a digital signature, and relies on hashing and public-key cryptography.

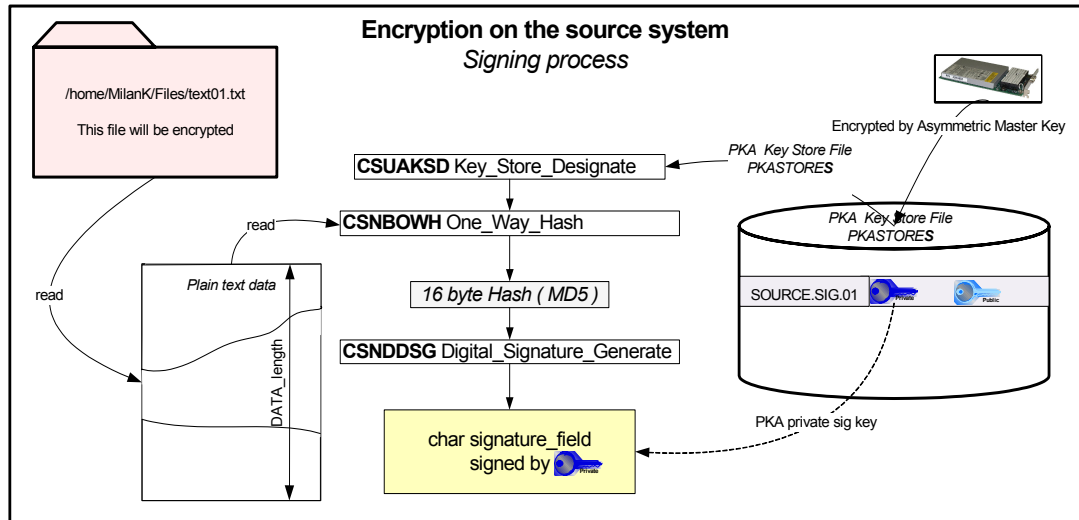


Figure 12-9 Signing process on the source system

The signing process is realized in two steps (Figure 12-9). Hash the text file `/home/MilanK/Files/text01.txt`. In our scenario we do not expect the length of the text file to be less than 64 bytes. The text file is specified as the first parameter of the program ENCDATA. We calculate a hash value (digest) from a text file string using the MD5 method by using the SAPI the One\_Way\_Hash CSNBOWH.

**Note:** The parameter describes the length of the hashing text on i5/OS systems for SAPI One\_Way\_Hash CSNBOW, and it is restricted to a maximum of 64 MB—64 bytes.

**Note:** If in SAPI CSNBOWH rule\_array FIRST or MIDDLE calls are made, the text size must be a multiple of the algorithm block size (64 bytes).

- For an encryption result of the hash we use our private signing key with key label SOURCE.SIG.01. This label is specified as the fifth parameter of the program ENCDATA. As a digital-signature-hash formatting method we use PKCS-1.1.

The SAPI CSNDDSG Digital\_Signature\_Generate verb is used to generate a digital signature. The encrypted hash value (digest) is called a digital signature.

- Encrypt the digital signature by the generated random DATA key.  
 Encrypting the digital signature is realized in two steps:
  - a. Generate a random double-length DATA key for the encryption of the digital signature, the current date, and the time, and simultaneously encipher it by the public encryption key TARGET.ENC.01 (Figure 12-10).

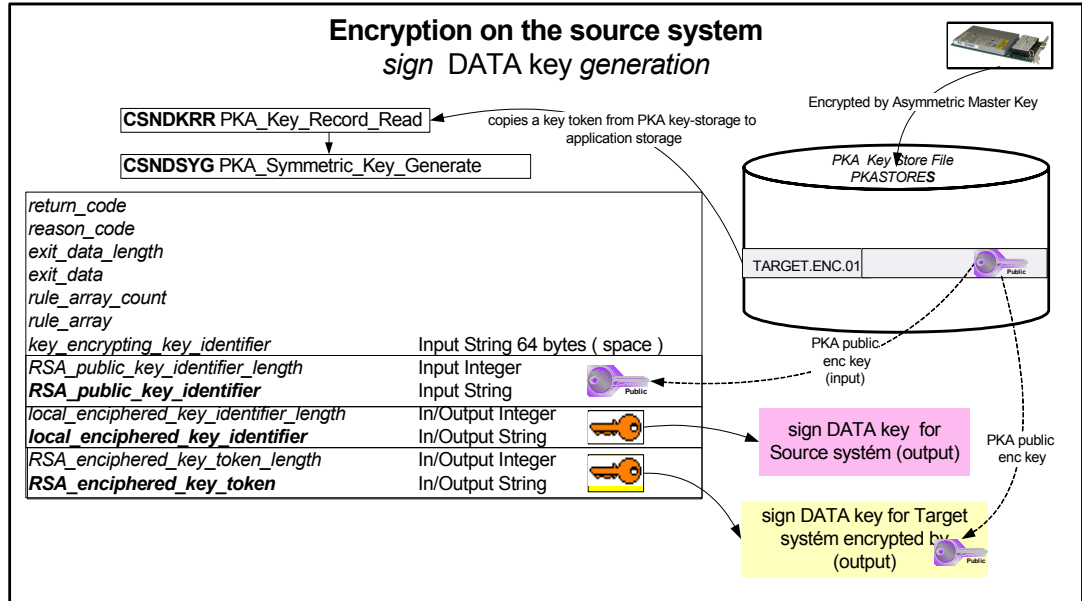


Figure 12-10 Generate a sign DATA key for the encryption of the digital signature and encipher it by the public encryption key

We use the SAPI CSNDKRR PKA\_Key\_Record\_Read verb to copy a key token from keystore file (PKA key-storage) PKASTORES to application storage. This key token is identified by key label TARGET.ENC.01 and presents the target public encryption key. This key token is used as an input parameter for encryption of a DATA key in the next sub-step (SAPI PKA\_Symmetric\_Key\_Generate verb CSNDSYG). The label of the public encryption key is the sixth parameter of the program ENCDATA.

Next we use the SAPI CSNDSYG PKA\_Symmetric\_Key\_Generate verb to generate a DATA key and to encipher it. This DATA key is enciphered under an RSA public-key. The key token of this RSA public-key was obtained in a previous sub-step and has the associated private key on the target system. This SAPI CSNDSYG returns two important parameters:

```
RSA_enciphered_key_token
local_enciphered_key_identifier
```

The first one containing the DATA key RSA-enciphered is used for decryption on the target system. The second one is used for the encryption on the source system in the next step. Both parameters must be sent to the target system.

- b. Encipher the digital signature, the date, and the time by the DATA key (local\_enciphered\_key\_identifier). The CSNBENC Encipher verb uses the DES algorithm and a secret generated by the DATA key to encipher data.



**Note:** Must we encipher the digital signature? No. But in some cases we want to send secret data and we do not mix in the encrypted text file (for example, the date and the time). We can add this data together with the digital signature and encipher it with the generated double length DATA key. In our scenario we add the time and the date of when the text file was encrypted (2007-08-17-09.33.32.6930).

Enciphered digital signing with its length, enciphered date, and the time of when text file was enciphered and information about the DATA key are stored in the stream file /home/MilanK/Files/text01.sig. The structure of this file is shown in Figure 12-11. The name of this file is specified as a second parameter of the program ENCDATA.

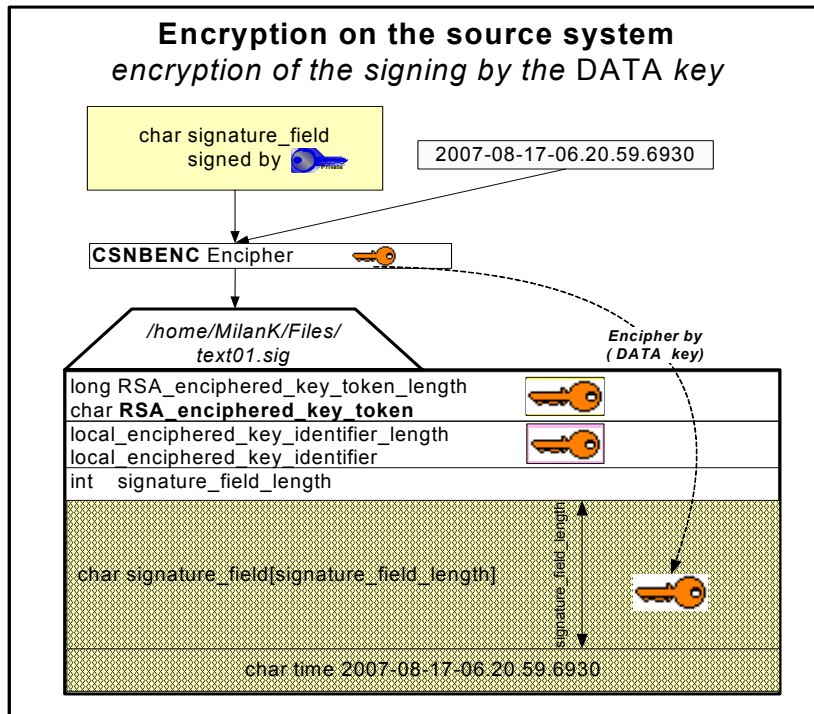


Figure 12-11 Encryption of signing by DES key

The structure information the DATA key and the length of the digital signature are the first part of the stream file /home/MilanK/Files/text01.sig and are shown in Example 12-7. The encrypted digital signature, the date, and the time are stored behind this information.

*Example 12-7 The structure of text01.sig and text01.enc file*

```
typedef _Packed struct file_Struc {
    long RSA_enciphered_key_token_length;
    char RSA_enciphered_key_token[2504];
    long local_enciphered_key_identifier_length;
    char local_enciphered_key_identifier[64];
    long field_length;
} file_Struc;
```

The parameter RSA\_enciphered\_key\_token describes a random DATA key enciphered under an RSA public-key TARGET.ENC.01. The parameter

local\_enciphered\_key\_identifier is used on the target system as information containing a control vector that conforms to the requirements of the key that is imported.

Both parameters are used on the target system in SAPI CSNDSYI PKA\_Symmetric\_Key\_Import for recovering a DATA key.

The field\_length parameter identifies the length of the encrypted digital signature. The length of the date and the time is known (24 bytes).

- ▶ Encrypt the data by the generated random DATA key.
- Encrypting the data is done in two steps:
- a. Generate a new random double length DATA key for the encryption of the data and simultaneously encipher it with the public encryption key TARGET.ENC.01. The two sub-steps are shown in Figure 12-12. These sub-steps are the same as the sub-steps for the digital signature. The only difference is the value of the random DATA key.

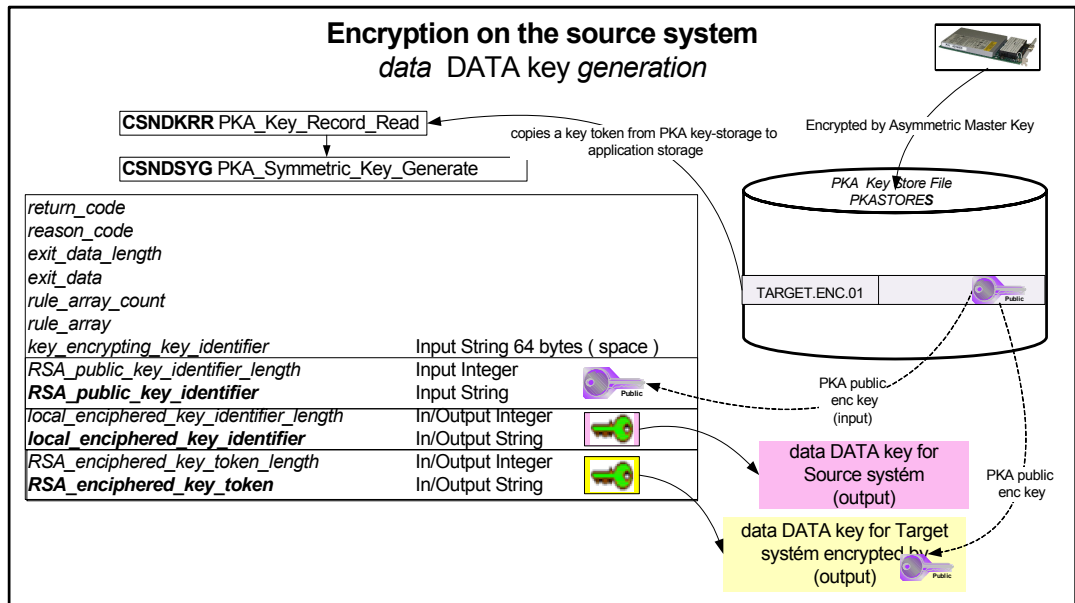


Figure 12-12 Generate a random double length DATA key for the encryption of the data and encipher it with the public encryption key

- b. Encipher the data with the generated random DATA key. We store all information in the stream file /home/MilanK/Files/text01.enc. The name of this file is specified as the third parameter of the program ENCDATA. Encrypting the data is shown in Figure 12-13.

**Note:** The parameter length on SAPI Encipher CSNBENC is restricted to a maximum of 64 MB (64 bytes) (for i5/OS systems).

**Note:** For ciphering method CBC, the data length must be a multiple of eight bytes.

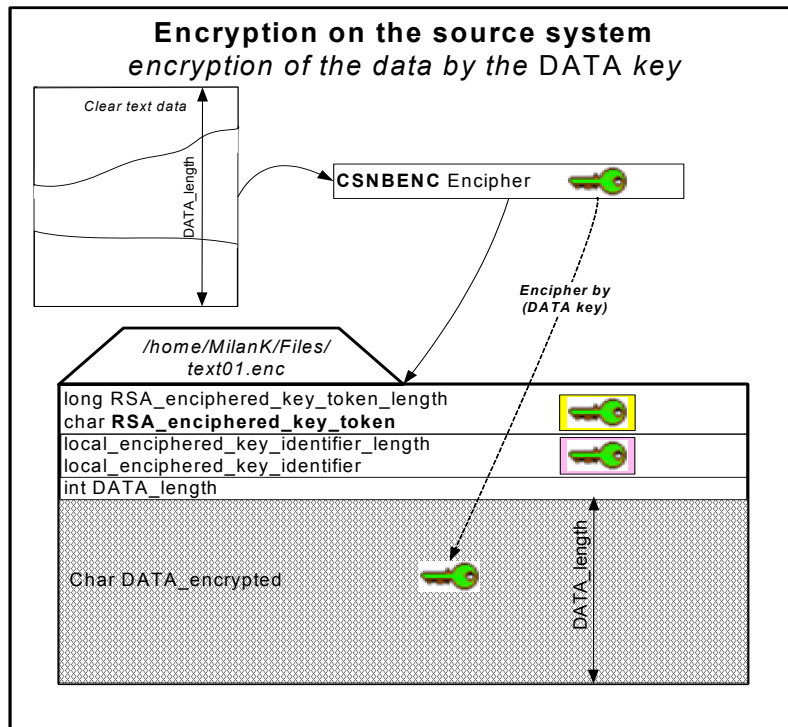


Figure 12-13 Encryption of data by DES key

This is the end of ENCDATA program description.

## Transferring encrypted files to target system

Now we transfer both stream files, text01.sig and text01.enc, to the target system.

For security reasons, in our environment we are not able to establish a direct connection between the source and the target system, so we have to transfer data between the source and the target system over our PC (Figure 12-5 on page 240).

To transfer stream files text01.sig and text01.enc from the source system RCH55 to our PC and from the PC to the target system Brs2as we use FTP commands (Example 12-8).

*Example 12-8 FTP commands to transfer stream encrypted files from the source to the target*

```
C:\>ftp RCH55
Connected to RCH55.
220-QTCP at RCHAS55.RCHLAND.IBM.COM.
220 Connection will close if idle more than 5 minutes.
```

```
User (RCH55:(none)): milank
331 Enter password.
Password:
230 MILANK logged on.
ftp> bin
200 Representation type is binary IMAGE.
ftp> quote site namefmt 1
250 Now using naming format "1".
ftp> lcd D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC
Local directory now D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC.
ftp> get /home/Milank/Files/text01.sig
200 PORT subcommand request successful.
150 Retrieving file /home/Milank/Files/text01.sig
226 File transfer completed successfully.
ftp: 2641 bytes received in 0,03Seconds 85,19Kbytes/sec.
ftp> get /home/Milank/Files/text01.enc
200 PORT subcommand request successful.
150 Retrieving file /home/Milank/Files/text01.enc
226 File transfer completed successfully.
ftp: 2761 bytes received in 0,00Seconds 2761000,00Kbytes/sec.
ftp> close
221 QUIT subcommand received.
ftp> open brs2as
Connected to brs2as.praha.cz.ibm.com.
220-QTCP at BRS2AS.
220 Connection will close if idle more than 5 minutes.
User (brs2as.praha.cz.ibm.com:(none)): cz50257
331 Enter password.
Password:
230 CZ50257 logged on.
ftp> bin
200 Representation type is binary IMAGE.
ftp> quote site namefmt 1
250 Now using naming format "1".
ftp> lcd D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC
Local directory now D:\RESIDENCY_2007_04_21\Finale_version\savf\CZ50257_CC.
ftp> cd /home/CZ50257/Files
250 "/home/CZ50257/Files" is current directory.
ftp> put text01.sig
200 PORT subcommand request successful.
150 Sending file to /home/CZ50257/Files/text01.sig
250 File transfer completed successfully.
ftp: 2641 bytes sent in 0,00Seconds 2641000,00Kbytes/sec.
ftp> put text01.enc
200 PORT subcommand request successful.
150 Sending file to /home/CZ50257/Files/text01.enc
250 File transfer completed successfully.
ftp: 2761 bytes sent in 0,00Seconds 2761000,00Kbytes/sec.
ftp> close
221 QUIT subcommand received.
ftp> quit
```

---

## Logging off and deallocating on source system

To do this:

1. Log off from our Cryptographic Coprocessor.

When we have finished with our Cryptographic Coprocessor, log off of it. We can log off by using the program LOGOFF that uses the Logon\_Control (CSUALCT) API verb. To start this program enter the command:

```
CALL PGM(LOGOFF)
```

**Note:** The program LOGOFF is included also in the manual *System i Networking Cryptographic hardware Version 5 Release 4*, available at:

<http://publib.boulder.ibm.com/infocenter/series/v5r4/topic/rzajc/rzajc.pdf>

2. Deallocate a Cryptographic Coprocessor device description from our job.

When we have finished using a Cryptographic Coprocessor, deallocate the Cryptographic Coprocessor by using the Cryptographic\_Resource\_Deallocate (CSUACRD) API verb. A cryptographic device description cannot be varied off until all jobs using the device have deallocated it.

To deallocate a Cryptographic Coprocessor device description to our job, enter the command:

```
CALL PGM(CRPDEALLOC) PARM(CRP02)
```

**Note:** The program CRPDEALLOC is included also in the manual *System i Networking Cryptographic hardware Version 5 Release 4*, available at:

<http://publib.boulder.ibm.com/infocenter/series/v5r4/topic/rzajc/rzajc.pdf>

## Adding public keys from source onto target system's keystore file

Now back to our target system. We add public keys from the transferred stream files from the source system to the keystore file. Transferred files Source.enc.01.pub and Source.sig.01.pub, including the public encryption and signing key, are added to the keystore file PKASTORET. The scheme is outlined in Figure 12-14.

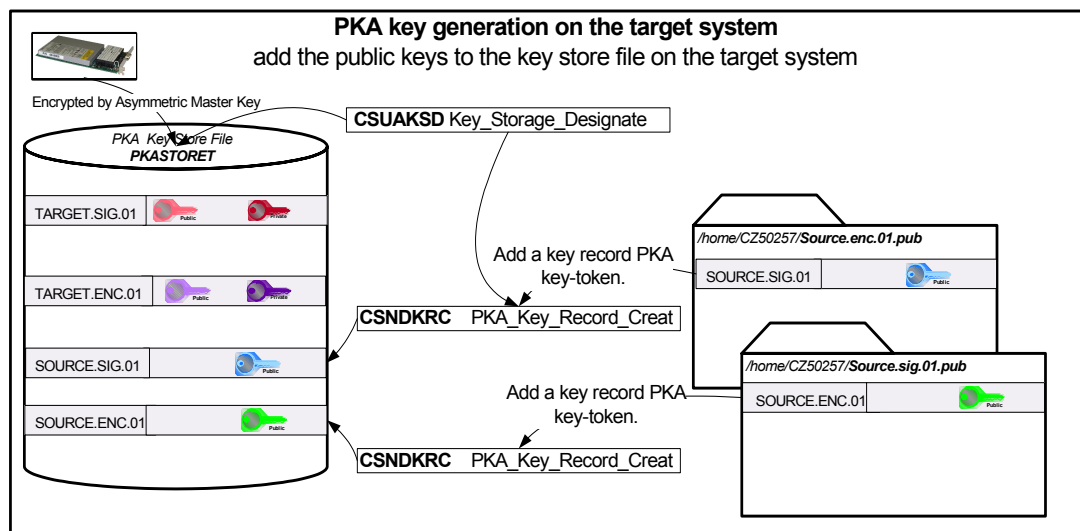


Figure 12-14 Add public keys to the keystore file on the target system

Run the following command:

```
CALL PGM(STRPUBKEY)
      PARM('PKASTORET R7399_CCA '
           '/home/CZ50257/Source.enc.01.pub '
           '/home/CZ50257/Source.sig.01.pub')
```

The parameters and their values used in this program execution are:

- ▶ PKASTORET R7399\_CCA - the fully qualified file name of the keystore file name
- ▶ /home/CZ50257/Source.enc.01.pub - extracted public key for encrypting
- ▶ /home/CZ50257/Source.sig.01.pub - extracted public key for signing

The program STRPUBKEY adds two records—two public keys transferred from the source system (Example 12-9).

*Example 12-9 QSYS/DSPPFM FILE(R7399\_CCA/PKASTORET) MBR(QAC6PKEY)*

---

```
Display Physical File Member
File . . . . . : PKASTORET          Library . . . . . : R7399_CCA
Member . . . . . : QAC6PKEY         Record . . . . . : 1
Control . . . . .                   Column . . . . . : 1
Find . . . . .
*...+...1...+...2...+...3...+...4...+...5...+...6...+...7..
TARGET ENC 01
TARGET SIG 01
SOURCE ENC 01
SOURCE SIG 01
***** END OF DATA *****
```

---

### **Description of STRPUBKEY program**

For the STRPUBKEY program:

- ▶ Designate the key-storage file.

For the job that ran our program, before we can perform any operation using the keystore file or keys stored in this keystore file, we must name the keystore file by the SAPI the Key\_Storage\_Designate CSUAKSD. The keystore file name R7399\_CCA/PKASTORET is specified as the first parameter to the program STRPUBKEY. In the next two steps we store the transferred public encryption/signing keys to this keystore file.

- ▶ Add the public encryption key to the keystore file name.

We retrieve information from the stream file Source.enc.01.pub specified as the second parameter of the program STRPUBKEY and use SAPI PKA\_Key\_Record\_Create CSNDKRC to store the public encryption key to the keystore file PKASTORET. The structure of the file Source.enc.01.pub is shown in Example 12-2 on page 239.

- ▶ Add the public signing key to the keystore file name.

We retrieve information from the stream file Source.sig.01.pub specified as the third parameter of the program STRPUBKEY and use SAPI PKA\_Key\_Record\_Create CSNDKRC to store the public signing key to the keystore file PKASTORET. The structure of the file Source.sig.01.pub is shown in Example 12-2 on page 239.

This is the end of STRPUBKEY program description.

### **Decrypting data**

Finally, we decrypt the data. We decrypt two stream files, which were the results of our encryption process. The first file holds an encrypted text file with the suffix <enc>. The second

file holds an encrypted digital signature of the text file, the date, and the time with the suffix <sig>.

Run the following command:

```
CALL PGM(DECDDATA)
      PARM('/home/CZ50257/Files/text01_dec.txt'
           '/home/CZ50257/Files/text01.sig'
           '/home/CZ50257/Files/text01.enc'
           'PKASTORET R7399_CCA '
           'SOURCE.SIG.01'
           'TARGET.ENC.01' )
```

The parameters and their values used in this program execution are:

- ▶ /home/CZ50257/Files/text01\_dec.txt: decrypted text file
- ▶ /home/CZ50257/Files/text01.sig: encrypted digital signature of the text file, the date, and the time
- ▶ /home/CZ50257/Files/text01.enc: encrypted text file
- ▶ PKASTORET R7399\_CCA: a fully qualified file name of the keystore file
- ▶ SOURCE.SIG.01: key label of the public key to verify digital signature
- ▶ TARGET.ENC.01: key label of the private key for encryption the DATA keys

### ***Description of DECDDATA program***

For the DECDDATA program:

- ▶ Designate the key-storage file.

For the job that ran our program, before we can perform any operation using the keystore file or keys stored in this keystore file, we must name the keystore file by the SAPI the Key\_Storage\_Designate CSUAKSD. The keystore file name R7399\_CCA/PKASTORET is specified as the fourth parameter of the program ENCDATA. In the next steps we use keys stored in this keystore file.

- ▶ Decrypt the digital signature, the date, and the time.
  - The decryption process has two steps:
    - a. Extract a random DATAkey for decryption of the encrypted digital signature, the date, and the time in the file /home/CZ50257/Files/text01.sig. The name of this file is specified as a second parameter of the program ENCDATA. The scheme is outlined in Figure 12-15.

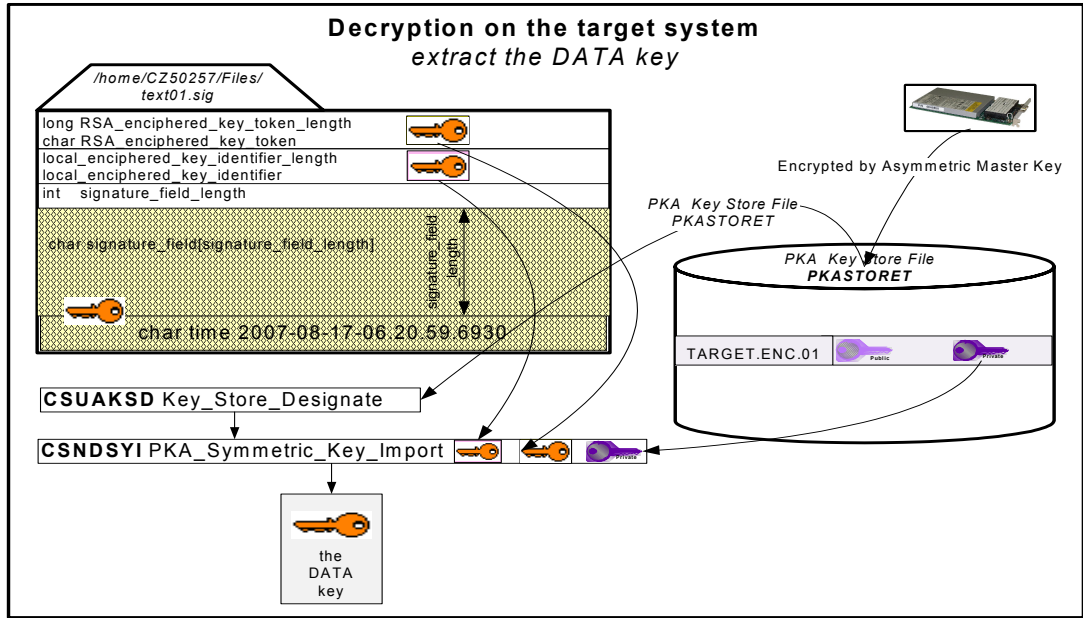


Figure 12-15 Extract the DATA key

The SAPI CSNDSYI PKA\_Symmetric\_Key\_Import verb decipheres the RSA-enciphered symmetric-key to be imported by using an RSA private-key, then multiply-enciphers the DATA key using the master key. The RSA private-key is identified by key label TARGET.ENC.01 and is specified as a sixth parameter of the program DECDATA. The multiply-enciphered DATA is used in SAPI CSNBDEC Decipher.



- b. Decipher the encrypted digital signature, the date, and the time with the imported DES symmetric-key. The Decipher verb uses the Data Encryption Standard (DES) algorithm and a cipher the DATA key obtained in previous step to decipher the digital signature, the date, and the time (Figure 12-16).

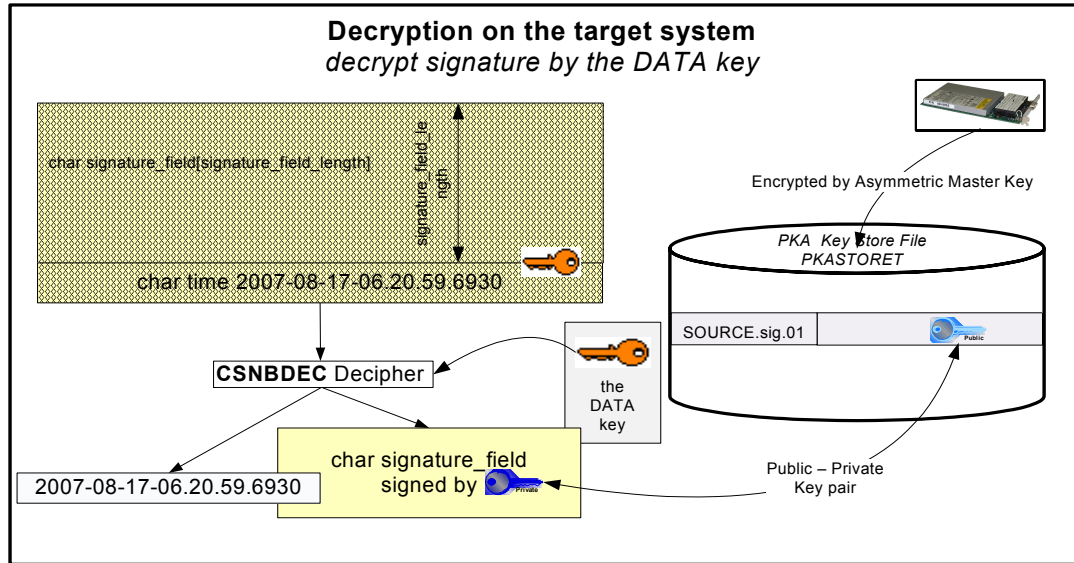


Figure 12-16 Decrypt signature by the DATA key

- Decrypt the data.

Decrypting the data has two steps:

- a. Extract a random DES key for decryption of the data in the file /home/CZ50257/Files/text01.enc. The name of this file is specified as a third parameter of the program ENCDATA. The scheme is outlined in Figure 12-17.

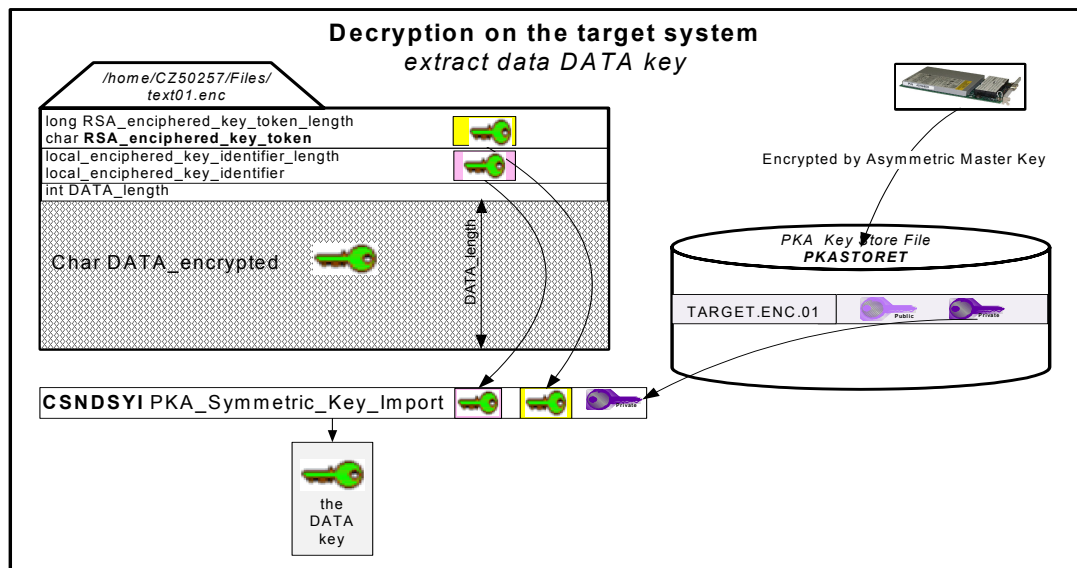


Figure 12-17 Extract data DATA key

The SAPI CSNDSYI PKA\_Symmetric\_Key\_Import verb deciphers the RSA-enciphered symmetric-key to be imported by using an RSA private-key, then multiply-enciphers

the data DATA key using the master key. The RSA private-key is identified by key label TARGET.ENC.01 and is specified as a sixth parameter of the program DECDATA. The multiply-enciphered DATA key is used in SAPI CSNBDEC Decipher.

- b. Decipher the data with the imported DATA key. The Decipher verb uses the Data Encryption Standard algorithm and a cipher DES key obtained in the previous step to decipher data (ciphertext). This verb results in data called plaintext. The name of this plaintext file is specified as a first parameter of the program DECDATA (/home/CZ50257/Files/text01\_dec.txt).

We use the random DATA key obtained in the previous sub-step to decrypt the data. The scheme is outlined in Figure 12-18. The encrypted data are in the stream file /home/CZ50257/Files/text01.enc. The name of this file is specified as a third parameter of the program ENCDATA.

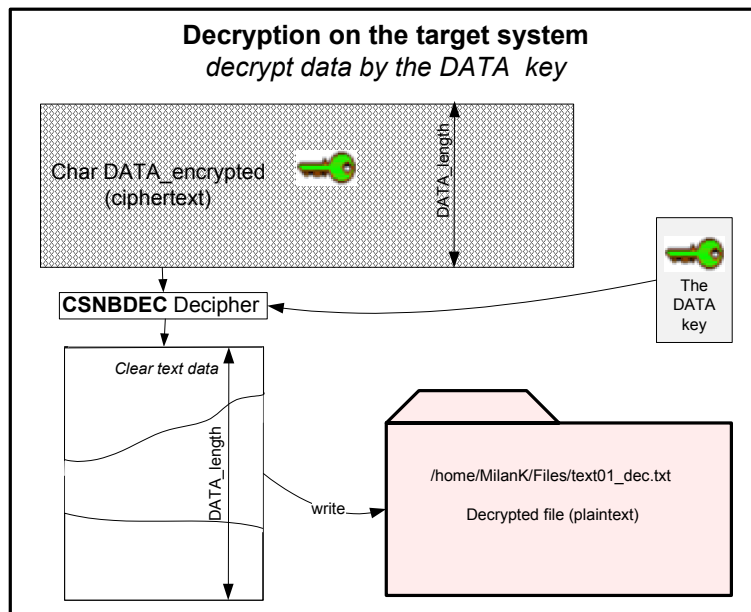


Figure 12-18 Decrypt data by the DATA key

► Verification process.

On the target system we have access to a public signing key SOURCE.SIG.01, and therefore we can verify the process as follows:

- a. Hash the data using the same hashing algorithm that we used to create the digital signature.
- b. Decrypt the digital signature using a public signing key (SOURCE.SIG.01).
- c. Compare the decrypted results to the hash value obtained from hashing the data.

An equal comparison confirms that the data that they possess is the same as that which we signed. The (CSNDDSG) Digital\_Signature\_Generate and the (CSNDDSV) Digital\_Signature\_Verify verbs perform the hash encrypting and decrypting operations.

At the end of this decryption we start the verification process to verify the digital signature. The scheme is outlined in Figure 12-19. The result of this verification process tell us whether the decrypted text file in the previous step /home/CZ50257/Files/text01\_dec.txt is the same as that which we signed.

The RSA public-key is identified by key label SOURCE.SIG.01 and is specified as a fifth parameter of the program DECDATA.

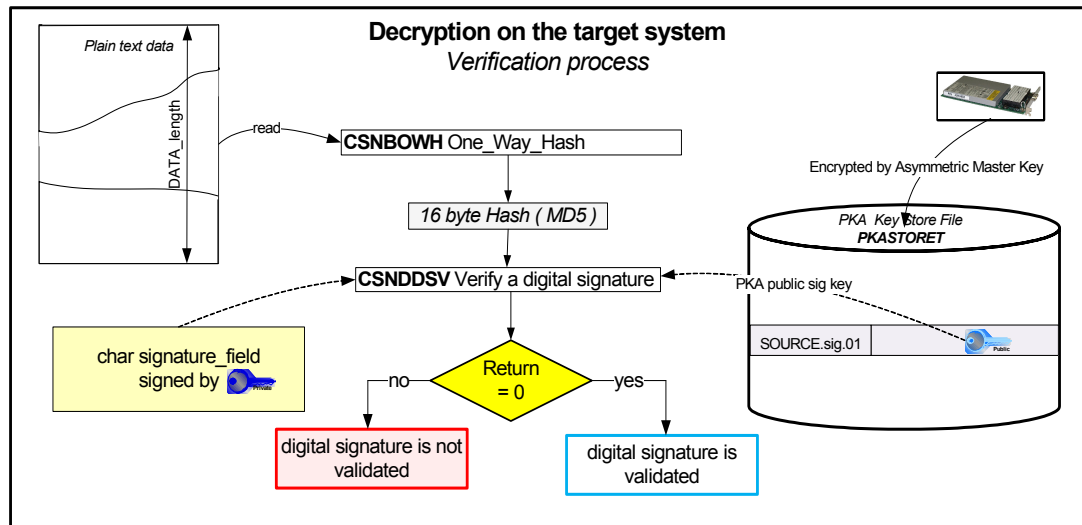


Figure 12-19 Verification process

This is the end of DECDATA program description.

## Logging off and deallocating on target system

To do this:

1. Log off from our Cryptographic Coprocessor.

When we have finished with our Cryptographic Coprocessor, log off of it. We can log off by using the program LOGOFF that uses the Logon\_Control (CSUALCT) API verb. To start this program we enter the command:

```
CALL PGM(LOGOFF)
```

**Note:** The program CRPDEALLOC is included also in the manual *System i Networking Cryptographic hardware Version 5 Release 4*, available at:

<http://publib.boulder.ibm.com/infocenter/iseres/v5r4/topic/rzajc/rzajc.pdf>

2. Deallocate a Cryptographic Coprocessor device description from our job.

When we have finished using a Cryptographic Coprocessor, we should deallocate the Cryptographic Coprocessor by using the Cryptographic\_Resource\_Deallocate (CSUACRD) API verb. A cryptographic device description cannot be varied off until all jobs using the device have deallocated it.

To deallocate a Cryptographic Coprocessor device description to our job, enter the command:

```
CALL PGM(CRPDEALLOC) PARM(CRP03)
```

**Note:** The program CRPDEALLOC is included also in the manual *System i Networking Cryptographic hardware Version 5 Release 4*, available at:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/rzajc/rzajc.pdf>

## 12.4.2 Execution example of scenario A

This section provides an execution example of scenario A. This might sound redundant, and it is in a way, but it gives you a more hands-on look and feel of the application along with actual messages that you might see. Consider this a summary application description with tagged along messages.

### **Environment setup**

To set up the environment:

1. Restore library R7399\_CCA from the downloaded save file to the source and to the source system.

**Note:** When we restore library R7399\_CCA, all programs needed for our scenario are in this library.

2. Compile the CL program CRTPGM on the source and on the target system:

```
CRTCLPGM PGM(R7399_CCA/CRTPGM)
          SRCFILE(R7399_CCA/QCLSRC)
          SRCMBR(CRTPGM)
          TEXT(*SRCMBRTXT) REPLACE(*YES) TGTRLS(*PRV)
```

3. To create all programs for our scenario A, start the program CRTPGM on the source and on the target system:

```
CALL PGM(R7399_CCA/CRTPGM)
```

4. Create a directory in the source system, where <user profile> is MilanK. In the target system <user profile> is CZ50257.

```
MKDIR DIR('/home/<user profile>')
MKDIR DIR('/home/<user profile>/Files')
```

5. Restore files that we want to encrypt to the directory /home/<user profile>/Files. In our scenarios we restore the files text01.txt and text02.txt.

On the source system RCH55 we encrypt the data text.01 and move encrypted data to the target system Brs2as where the data will be decrypted. On the Cryptocard coprocessor CRP02 on the source system and on the Cryptocard coprocessor CRP03 on the target system, we created profile ALL with roles all in which we enabled all roles that have the access control point that we want to use.

6. On the source system we perform the following steps:
  - a. Specify the library R7399\_CCA to be added to the user portion of the library list for our job:

```
ADDLIBLE LIB(R7399_CCA)
```

- b. Allocate the Cryptographic Coprocessor device description to our job:

```
CALL PGM(CRPALLOC) PARM(CRP02)
Request was successful
Press ENTER to end terminal session.
```

- c. Log onto our Cryptographic Coprocessor:

```
CALL PGM(LOGON) PARM('ALL' 'all')
Logon was successful
Press ENTER to end terminal session.
```

- d. Generate two asymmetrical key pairs and store both pairs keys (private,public) to the keystore file and extract the public keys from this keystore file. Store them to the stream files:

```
CALL PGM(PKAKEYGEN)
  PARM('*YES'
        'SOURCE.ENC.01'
        'SOURCE.SIG.01'
        'PKASTORES R7399_CCA '
        '*YES'
        '/home/MilanK/Source.enc.01.pub'
        '/home/MilanK/Source.sig.01.pub')
```

```
>>> PKAKEYGEN <<<
Key store file R7399_CCA/PKASTORES created

Key store designated
SAPI returned 0/0

ENC Record added to key store
SAPI returned 0/0

ENC Key token built
ENC Key generated and stored in key store
SAPI returned 0/0

ENC Public Key extracted from key store
SAPI returned 0/0

ENC Public Key extracted was written to file

SIG Record added to key store
SAPI returned 0/0

SIG Key token built
SIG Key generated and stored in key store
SAPI returned 0/0

SIG Public Key extracted from key store
SAPI returned 0/0

SIG Public Key extracted was written to file
Press ENTER to end terminal session.
```

- e. Transfer two stream files with the public key to the target system (see Example 12-3 on page 240):

```
/home/MilanK/Source.enc.01.pub
/home/MilanK/Source.sig.01.pub
```

7. On the target system we have to perform the following steps:
- Specify the library R7399\_CCA to be added to the user portion of the library list for our job:

```
ADDLIBLE R7399_CCA
```

- Allocate a Cryptographic Coprocessor device description to our job:

```
CALL PGM(CRPALLOC) PARM(CRP03)
Request was successful
Press ENTER to end terminal session.
```

- Log on into our Cryptographic Coprocessor:

```
CALL PGM(LOGON) PARM('ALL' 'all')
Logon was successful
Press ENTER to end terminal session.
```

- Generate two asymmetrical key pairs and store both pairs keys (private,public) to the keystore file, and extract the public keys from this keystore file. Store them to the stream files:

```
CALL PGM(PKAKEYGEN)
  PARM('*YES'
        'TARGET.ENC.01'
        'TARGET.SIG.01'
        'PKASTORET R7399_CCA '
        '*YES'
        '/home/CZ50257/Target.enc.01.pub'
        '/home/CZ50257/Target.sig.01.pub')
```

```
>>> PKAKEYGEN <<<
Key store file R7399_CCA/PKASTORET created
```

```
Key store designated
SAPI returned 0/0
```

```
ENC Record added to key store
SAPI returned 0/0
```

```
ENC Key token built
ENC Key generated and stored in key store
SAPI returned 0/0
```

```
ENC Public Key extracted from key store
SAPI returned 0/0
```

```
ENC Public Key extracted was written to file
```

```
SIG Record added to key store
SAPI returned 0/0
```

```
SIG Key token built
SIG Key generated and stored in key store
SAPI returned 0/0
```

```
SIG Public Key extracted from key store
SAPI returned 0/0
```

```
SIG Public Key extracted was written to file
Press ENTER to end terminal session.
```

- e. Transfer two stream files with the public key to the source system (see Example 12-5 on page 245):

```
/home/CZ50257/Target.enc.01.pub
/home/CZ50257/Target.sig.01.pub
```

- 8. Next on the source system we perform the following steps:

- a. Add public keys from the transferred stream files from the target system to the keystore file PKASTORES:

```
CALL PGM(STRPUBKEY)
      PARM('PKASTORES R7399_CCA '
           '/home/MilanK/Target.enc.01.pub'
           '/home/MilanK/Target.sig.01.pub')
```

```
>>> STRPUBKEY <<<
Key store designated
SAPI returned 0/0
```

```
ENC Record added to key store
SAPI returned 0/0
```

```
SIG Record added to key store
SAPI returned 0/0
```

```
Press ENTER to end terminal session.
```

- b. Display text data text01.txt:

```
DSPF STMF('/home/MilanK/Files/text01.txt')
*****Beginning of data*****
*****
0123456789
1234567890
2345678901
3456789012
4567890123
5678901234
6789012345
7890123456
8901234567
*****
* text01 *
*****
*****End of Data*****
```

- c. Encrypt data text01.txt:

```
CALL PGM(ENCDATA)
      PARM('/home/MilanK/Files/text01.txt'
           '/home/MilanK/Files/text01.sig'
           '/home/MilanK/Files/text01.enc'
           'PKASTORES R7399_CCA '
           'SOURCE.SIG.01'
           'TARGET.ENC.01' )
```

```
>>> ENCDATA <<< 2007-09-15-14.00.49.5580
Key store designated
SAPI returned 0/0
```

```
Hash completed successfully.
SAPI returned 0/0
```

```
Signature generation was successful
Signature has length = 64
SAPI returned 0/0
```

```
ENC Public Key extracted from key store
SAPI returned 0/0
```

```
Random DES-key generated successfully
SAPI returned 0/0
```

```
Signature enciphered successfully
SAPI returned 0/0
```

```
Random DES-key generated successfully
SAPI returned 0/0
```

```
DATA enciphering completed successfully.
SAPI returned 0/0
```

Press ENTER to end terminal session.

- d. Transfer both stream files text01.sig and text01.enc to the target system (Example 12-8 on page 253).

- e. Log off from our Cryptographic Coprocessor:

```
CALL PGM(LOGOFF)
Log off successful
Press ENTER to end terminal session.
```

- f. Deallocate a Cryptographic Coprocessor device description from our job:

```
CALL PGM(CRPDEALLOC) PARM(CRP02)
Request was successful
Press ENTER to end terminal session.
```

- 9. Next on the target system we perform the following steps:

- a. Add public keys from the transferred stream files from the source system to the keystore file PKASTORET:

```
CALL PGM(STRPUBKEY)
  PARM('PKASTORET R7399_CCA '
        '/home/CZ50257/Source.enc.01.pub'
        '/home/CZ50257/Source.sig.01.pub')
```

```
>>> STRPUBKEY <<<
Key store designated
SAPI returned 0/0
```

```
ENC Record added to key store
SAPI returned 0/0
```



```
SIG Record added to key store
SAPI returned 0/0
```

Press ENTER to end terminal session.

- b. Decrypt the data and store to text01\_dec.txt:

```
CALL PGM(DECADATA)
      PARM('/home/CZ50257/Files/text01_dec.txt'
           '/home/CZ50257/Files/text01.sig'
           '/home/CZ50257/Files/text01.enc'
           'PKASTORET R7399_CCA '
           'SOURCE.SIG.01'
           'TARGET.ENC.01' )
```

```
>>> DECADATA <<<
Key store designated
SAPI returned 0/0
Symetric key Import successful
SAPI returned 0/0
```

```
Decipher signature was succesfull 2007-09-15-14.00.49.5580
SAPI returned 0/0
```

```
Symetric key Import successful
SAPI returned 0/0
```

```
DATA deciphering completed successfully.
SAPI returned 0/0
```

```
Hash completed successfully.
SAPI returned 0/0
```

```
Signature verification was successful.Return/Reason codes = 0/0
Press ENTER to end terminal session.
```

- c. Display decrypted data text01\_dec.txt:

```
DSPF STMF('/home/MilanK/Files/text01_dec.txt')
*****Beginning of data*****
*****
0123456789
1234567890
2345678901
3456789012
4567890123
5678901234
6789012345
7890123456
8901234567
*****
* text01 *
*****
*****End of Data*****
```

- d. Log off from our Cryptographic Coprocessor:

```
CALL PGM(LOGOFF)
Log off successful
```

Press ENTER to end terminal session.

- e. Deallocate a Cryptographic Coprocessor device description from our job:

```
CALL PGM(CRPDEALLOC) PARM(CRP03)
Request was successful
Press ENTER to end terminal session.
```

## 12.5 Data encryption/decryption on same system (scenario B)

In this scenario we have one system with Cryptographic Coprocessor 4758/4764 and we want to encrypt/decrypt data only on this system. For a scenario description, refer to 12.1.2, “Scenario B: encryption/decryption of data on the same system” on page 232.

### Process summary

This section highlights the tasks to be performed in this scenario:

- ▶ System setup and keys generation
- ▶ Encrypting data
- ▶ Decrypting data
- ▶ Logging off and deallocating

### 12.5.1 Single system scenario: step-by-step guide

This section provides detailed information about each step summarized in the previous section.

#### System setup and keys generation

To do this:

1. Specify the library R7399\_CCA to be added to the user portion of the library list for our job:

```
ADDLIBLE LIB(R7399_CCA)
```

2. Allocate a Cryptographic Coprocessor device description to our job:

```
CALL PGM(CRPALLOC) PARM(CRP02)
```

This program uses the Cryptographic\_Resource\_Allocate (CSUACRA) API verb to explicitly allocate a cryptographic device to our job so that the system can determine how to route all subsequent cryptographic requests.

If we use any of the CCA API verbs without first explicitly using the Cryptographic\_Resource\_Allocate (CSUACRA) API verb, the system will attempt to allocate the default cryptographic device. The default device is the cryptographic device named CRP01. It must be created by either using the Basic Configuration wizard or the Create Device Crypto (CRTDEVCRP) CL command.

We only need to use CSUACRA when we wish to use a device other than the default cryptographic device. A device allocated to a job, either explicitly or implicitly, remains allocated until either the job ends or the device is deallocated using the Cryptographic\_Resource\_Deallocate (CSUACRD) API verb.

**Note:** The program CRPALLOC is included also in the manual *System i Networking Cryptographic hardware Version 5 Release 4*, available at:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/topic/rzajc/rzajc.pdf>

3. Log onto the Cryptographic Coprocessor.

We can log on by using the program LOGON that uses the Logon\_Control (CSUALCT) API verb. To start this program we enter the command:

```
CALL PGM(LOGON) PARM('user ID' 'passphrase')
```

Where 'user ID' is string that identifies the user to the system and 'passphrase' is data used in the authentication process.

We need to log on only if we wish to use APIs that use access control points that are not enabled in the default role. Log on with a profile that uses a role that has the access control point that we want to use enabled.

**Note:** The program LOGON is included also in the manual *System i Networking Cryptographic hardware Version 5 Release 4*, available at:

<http://publib.boulder.ibm.com/infocenter/series/v5r4/topic/rzajc/rzajc.pdf>

4. Generate two asymmetric key pairs.

Generate one pair of asymmetric keys (private, public with the label SOURCE.SIG.01) for the signing and verification process and one pair of asymmetric keys (private,public with the label SOURCE.ENC.01) for encrypting and decrypting data, and store both key pairs in the keystore file PKASTORES.

Run the following command:

```
CALL PGM(PKAKEYGEN)
  PARM('*YES '
        'SOURCE.ENC.01 '
        'SOURCE.SIG.01 '
        'PKASTORES R7399_CCA '
        '*NO ')
```

The parameters and their values in this program execution are:

- \*YES: A keystore file, R7399\_CCA/PKASTORES, is to be initialized.
- SOURCE.ENC.01: key label of asymmetrical key pair for encryption.
- SOURCE.SIG.01: key label of asymmetrical key pair for signing.
- PKASTORES R7399\_CCA: a fully qualified file name of the keystore file.
- \*NO: This is not to extract a public key from the keystore file.

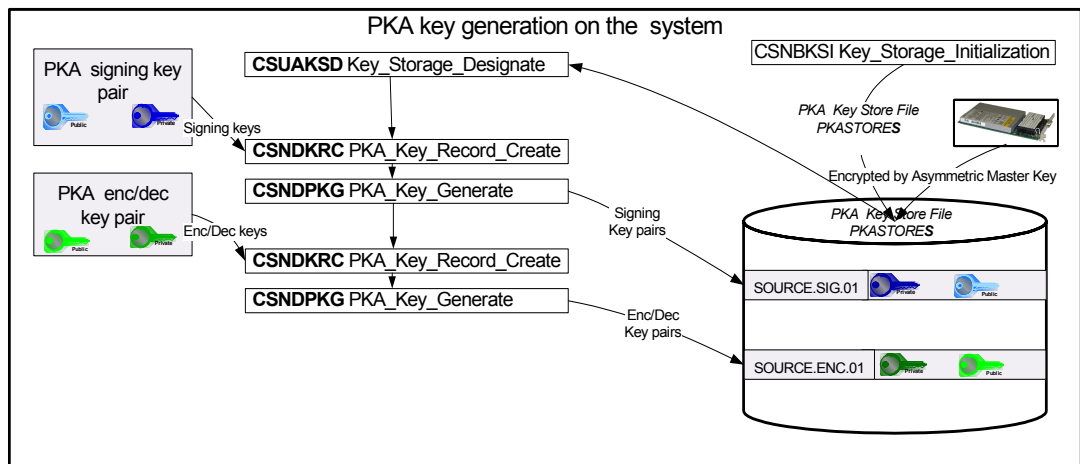


Figure 12-20 Generation of the asymmetrical key pairs on the system

The program PKAKEYGEN creates in key storage file (PKA key-store file) PKASTORES two records—two key pairs of asymmetric keys (Example 12-1 on page 238).

### **What PKAKEYGEN performs**

For a detailed description of the PKAKEYGEN program, refer to “Description of PKAKEYGEN program” on page 238.

The only difference from how this program acted for scenario A and here is that we do not need an extracted public key for encryption and signing (files Source.enc.01.pub and Source.sig.01.pub) because the encrypting and decrypting process is enacted on the same system. The keystore file PKASTORES includes all keys needed (Figure 12-20 on page 269):

- ▶ SOURCE.ENC.01: public private keys for encryption/decryption
- ▶ SOURCE.SIG.01: public private key for signing/verification

### **Encrypting data**

We encrypt stream text file 'text01.txt'. As a result of our encryption process we have two files. The first file holds an encrypted text file. This file has the suffix <enc>, as in 'text01.enc'. The second file holds the encrypted digital signature of the text file, the date, and the time. This file has the suffix <sig>, as in 'text01.sig'.

For encryption we use a generated random double length DATA key. This DATA key is saved together with the encrypted text file, the digital signature, the date, and the time. The DATA key is in an encrypt form. It is encrypted by an asymmetric public key (SOURCE.ENC.01). The structures of these files are shown in Figure 12-11 on page 251 and Figure 12-13 on page 253.

Run the following command:

```
CALL PGM(ENCDATA)
      PARM('/home/MilanK/Files/text01.txt'
           '/home/MilanK/Files/text01.sig'
           '/home/MilanK/Files/text01.enc'
           'PKASTORES R7399_CCA '
           'SOURCE.SIG.01'
           'SOURCE.ENC.01' )
```

The parameters and their values in this program execution are:

- ▶ /home/MilanK/Files/text01.txt: a text file to be encrypted
- ▶ /home/MilanK/Files/text01.sig: an encrypted digital signature of the text file, the date, and the time
- ▶ /home/MilanK/Files/text01.enc: an encrypted text file
- ▶ PKASTORES R7399\_CCA: a fully qualified file name of the keystore file
- ▶ SOURCE.SIG.01: key label of the private key for signing the digest of the text file
- ▶ SOURCE.ENC.01: key label of the public key for encryption the DATA keys

### **Description of ENCDATA program**

For the ENCDATA program:

1. Designate the key-storage file.

For the job that ran our program before we can perform any operation using the keystore file or keys stored in this keystore file, we must name the keystore file by the SAPI the Key\_Storage\_Designate CSUAKSD. The keystore file named R7399\_CCA/PKASTORES

is specified as the fourth parameter of the program ENCDATA. In the next steps we use keys stored in this keystore file.

2. Retrieve the current date and the time and generate the digital signature of the data that will be encrypted.

We retrieve the current date and the time and calculate the digital signature of the text file (Figure 12-9 on page 249). This step is the same as steps in scenario A.

3. Generate a random double length DATA key for the encryption of the digital signature, the date, and the time, and simultaneously encipher it with the public key (SOURCE.ENC.01).  
This steps are almost the same as steps in scenario A (Figure 12-21).

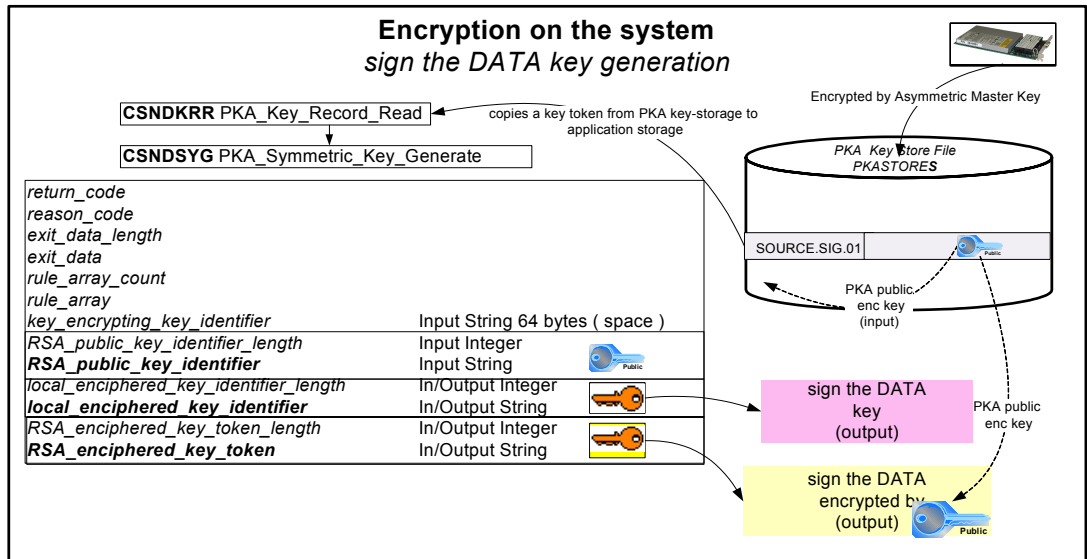


Figure 12-21 Sign the DATA key generation on the system

For encryption of the DATA key we use now SOURCE.ENC.01,

4. Encrypt the digital signature, the date, and the time with the generated DATA key. Store these encrypted data, the length of the enciphered digital signature, and the enciphered DATA key to the stream file (text01.sig).

These steps are the same as in scenario A (Figure 12-11 on page 251).

5. Generate a new random double length DATA key for the encryption of the data and simultaneously encipher it with the public key (SOURCE.ENC.01), as shown in Figure 12-22.

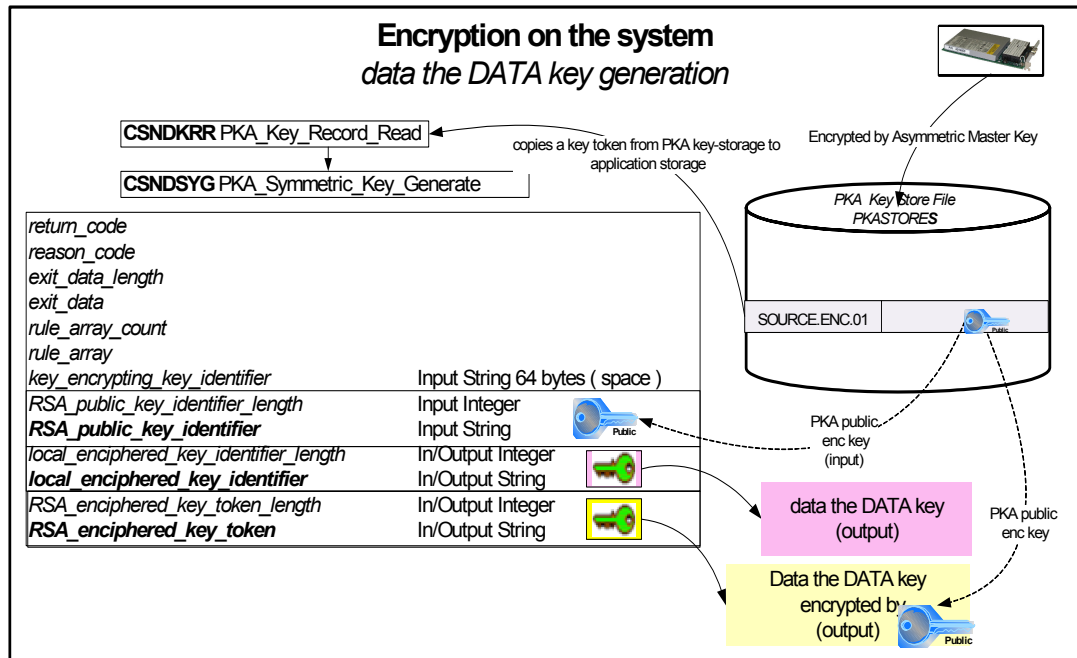


Figure 12-22 The DATA key generation on the system

For encryption of the DATA key we use SOURCE.ENC.01.

6. Encrypt the data with the generated random double length DATA key. Store encrypted data and its length with the enciphered DATA key to the stream file (text01.enc).

These steps are the same as in scenario A (Figure 12-13 on page 253).

This is the end of ENCDATA program description.

## Decrypting data

We decrypt two stream files. These two files are a result of our encryption process. The first one is the file includes the encrypted text file with the suffix enc. The second one includes an encrypted digital signature of the text file, the date, and the time with the suffix sig.

Run the following command:

```
CALL PGM(DECDDATA)
  PARM('/home/MilanK/Files/text01_dec.txt'
        '/home/MilanK/Files/text01.sig'
        '/home/MilanK/Files/text01.enc'
        'PKASTORES R7399_CCA '
        'SOURCE.SIG.01'
        'SOURCE.ENC.01' )
```

The parameters and their values in this program execution are:

- ▶ /home/CZ50257/Files/text01\_dec.txt: a decrypted text file
- ▶ /home/CZ50257/Files/text01.sig: an encrypted digital signature of the text file, the date, and the time

- ▶ /home/CZ50257/Files/text01.enc: an encrypted text file
- ▶ PKASTORET R7399\_CCA: a fully qualified file name of the keystore file
- ▶ SOURCE.SIG.01: key label of the public key for Verification digital signature
- ▶ SOURCE.ENC.01: key label of the private key for encryption the DATA keys

**Description of DECDATA program**

For the DECDATA program:

1. Designate the key-storage file.

For the job that ran our program, before we can perform any operation using the keystore file or keys stored in this keystore file, we must name the keystore file by the SAPI the Key\_Storage\_Designate CSUAKSD. The keystore file named R7399\_CCA/PKASTORES is specified as the fourth parameter of the program ENCDATA. In the next steps we use keys stored in this keystore file.

2. Decrypt the digital signature.

Decryption the digital signature is done in two steps:

- a. Extract the DATA key for decryption of the encrypted digital signature, the date, and the time in the file /home/CZ50257/Files/text01.sig. The name of this file is specified as a second parameter of the program ENCDATA (Figure 12-23).

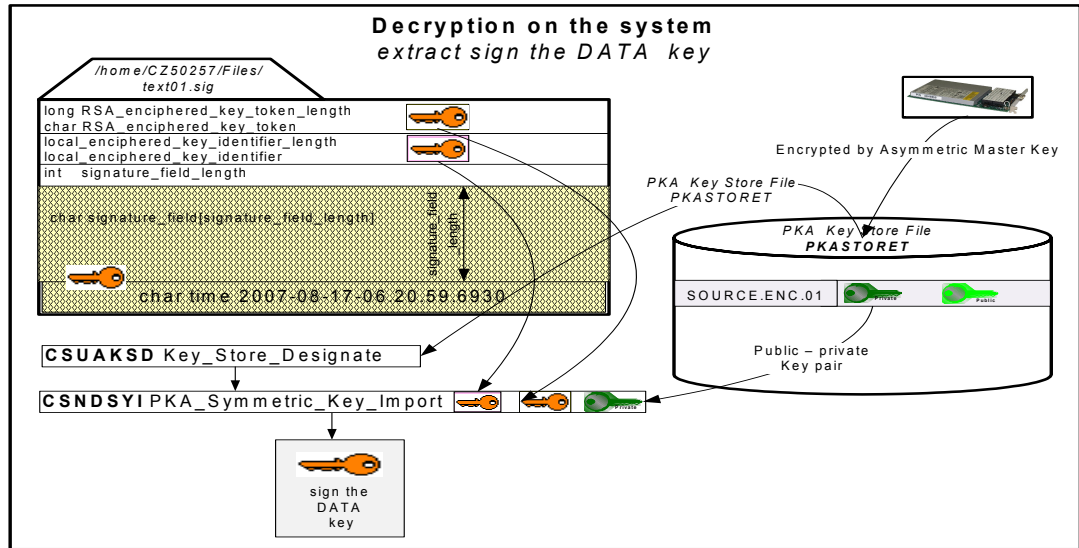


Figure 12-23 Extract sign the DATA key on the system

The SAPI CSNDSYI PKA\_Symmetric\_Key\_Import verb deciphers the RSA-enciphered symmetric-key to be imported by using an RSA private-key, then multiply-enciphers the DATA using the master key. The RSA private-key is identified by key label SOURCE.ENC.01, and it is specified as a sixth parameter of the program DECDATA. The multiply-enciphered DATA key is used in SAPI CSNBDEC Decipher.

- b. Decipher the encrypted digital signature, the date, and the time with the imported DES symmetric-key. The Decipher verb uses the Data Encryption Standard algorithm and a cipher the DATA key obtained in a previous step to decipher the digital signature, the date, and the time (Figure 12-16 on page 259).

3. Decrypt the data.

Decrypting the data is done in two steps:

- a. Extract the DATA key for decrypting the data in the file `/home/CZ50257/Files/text01.enc`. The name of this file is specified as a third parameter of the program ENCDATA. The scheme is outlined in Figure 12-24.

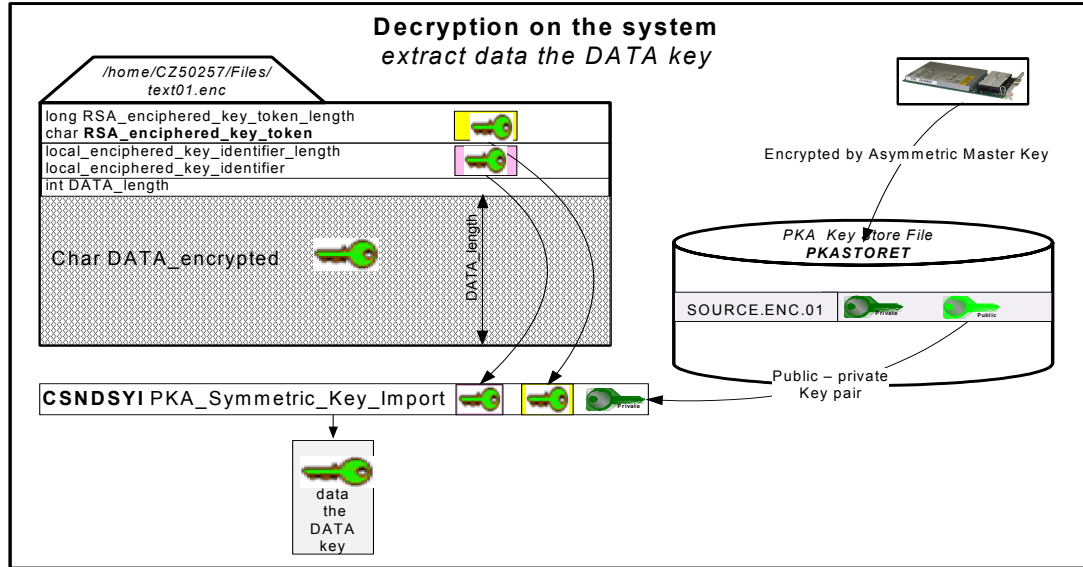


Figure 12-24 Extract data the DATA key on the system

The SAPI CSNDSYI PKA\_Symmetric\_Key\_Import verb decipheres the RSA-enciphered symmetric-key to be imported by using an RSA private-key, then multiply-enciphers the DATA key using the master key. The RSA private-key is identified by key label SOURCE.ENC.01 and is specified as a sixth parameter of the program DECDATA. The multiply-enciphered DATA is used in SAPI CSNBDEC Decipher.

- b. Decipher the data by importing the DATA key. The Decipher verb uses the Data Encryption Standard algorithm and a cipher DES key obtained in a previous step to decipher data (ciphertext). This verb results in data called plaintext. The name of this plaintext file `/home/CZ50257/Files/text01_dec.txt` is specified as a first parameter of the program DECDATA.

We use the DATA key obtained in a previous sub-step to decrypt the data. The scheme is the same as in Figure 12-18 on page 260—decrypting on the target system. The encrypted data are in the stream file `/home/CZ50257/Files/text01.enc`, the third parameter of program DECDATA.

4. Verification process.

On the system we have access to a public signing key SOURCE.SIG.01, and therefore we can verify information as follows:

- a. Hash the data using the same hashing algorithm that we used to create the digital signature.
- b. Decrypt the digital signature using a public signing key.
- c. Compare the decrypted results to the hash value obtained from hashing the data.

An equal comparison confirms that the data that they possess is the same as that which we signed. The (CSNDDSG) Digital\_Signature\_Generate and the (CSNDDSV) Digital\_Signature\_Verify verbs perform the hash encrypting and decrypting operations.



At the end of this decryption we start the verification process to verify the digital signature. The scheme is outlined in Figure 12-19 on page 261. The result of this verification process tell us whether the decrypted text file in the previous step /home/CZ50257/Files/text01\_dec.txt is the same as that which we signed.

This is the end of DECDATA program description.

## Logging off and deallocating

When we have finished with our Cryptographic Coprocessor, log off of it.

1. We can log off by using the program LOGOFF that uses the Logon\_Control (CSUALCT) API verb. To start this program we enter the command:

```
CALL PGM(LOGOFF)
```

**Note:** The program LOGOFF is included also in the manual *System i Networking Cryptographic hardware Version 5 Release 4*, available at:

<http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/rzajc/rzajc.pdf>

2. Deallocate a Cryptographic Coprocessor device description from our job.

When we have finished using a Cryptographic Coprocessor, we should deallocate the Cryptographic Coprocessor by using the Cryptographic\_Resource\_Deallocate (CSUACRD) API verb. A cryptographic device description cannot be varied off until all jobs using the device have deallocated it.

To deallocate a Cryptographic Coprocessor device description to our job, we need to enter the command:

```
CALL PGM(CRPDEALLOC) PARM(CRP01)
```

**Note:** The program CRPDEALLOC is included also in the manual *System i Networking Cryptographic hardware Version 5 Release 4*, available at:

<http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/rzajc/rzajc.pdf>

## 12.5.2 Execution example of scenario B

This section provides an execution example of scenario B. This might sound redundant, and it is in a way, but it gives you more a hands-on look at the application, along with actual messages you might see. Consider this as a summary of application description with tagged along messages.

### **Environment setup**

To set up the environment:

1. Restore library R7399\_CCA from the downloaded save file to the source system.
2. Compile the CL program CRTPGM on the source system:

```
CRTCLPGM PGM(R7399_CCA/CRTPGM)
         SRCFILE(R7399_CCA/QCLSRC)
         SRCMBR(CRTPGM)
         TEXT(*SRCMBRTXT) REPLACE(*YES) TGTRLS(*PRV)
```

3. To create all programs for our scenario B, we need to start the program CRTPGM on the source system:

```
CALL PGM(R7399_CCA/CRTPGM)
```

4. Create a directory in the source system. <user profile> is MilanK:

```
MKDIR DIR('/home/<user profile>')
MKDIR DIR('/home/<user profilr>/Files')
```

5. Restore files that we want to encrypt to the directory /home/<user profile>/Files. In our scenarios we restore the files text01.txt and text02.txt.

Now we encrypt data text.01 on the source system RCH55. On the Cryptocard coprocessor CRP02 we created profile ALL with role all in which we enabled all roles that have the access control point that we want to use.

6. Specify the library R7399\_CCA to be added to the user portion of the library list for the our job:

```
ADDLIB LIB(R7399_CCA)
```

7. Allocate a Cryptographic Coprocessor device description to our job:

```
CALL PGM(CRPALLOC) PARM(CRP02)
Request was successful
Press ENTER to end terminal session.
```

8. Log on into our Cryptographic Coprocessor:

```
CALL PGM(LOGON) PARM('ALL' 'all')
Logon was successful
Press ENTER to end terminal session.
```

9. Generate two asymmetric key pairs:

```
CALL PGM(PKAKEYGEN)
  PARM('*YES'
        'SOURCE.ENC.01'
        'SOURCE.SIG.01'
        'PKASTORES R7399_CCA '
        '*NO ' )
>>> PKAKEYGEN <<<
Key store file R7399_CCA/PKASTORES created
```

```
Key store designated
SAPI returned 0/0
```

```
ENC Record added to key store
SAPI returned 0/0
```

```
ENC Key token built
ENC Key generated and stored in key store
SAPI returned 0/0
SIG Record added to key store
SAPI returned 0/0
```

```
SIG Key token built
SIG Key generated and stored in key store
SAPI returned 0/0
```

```
Press ENTER to end terminal session.
```

10. Encrypt data.

- a. Display text data (plaintext):

```
DSPF STMF('/home/MilanK/Files/text01.txt')
```

```

*****Beginning of data*****
*****
0123456789
1234567890
2345678901
3456789012
4567890123
5678901234
6789012345
7890123456
8901234567
*****
* text01 *
*****
*****End of Data*****

```

- b. Encrypt data, generate ciphertext text01.sig and text01.enc.

```

CALL PGM(ENCDATA)
      PARM('/home/MilanK/Files/text01.txt'
           '/home/MilanK/Files/text01.sig'
           '/home/MilanK/Files/text01.enc'
           'PKASTORES R7399_CCA '
           'SOURCE.SIG.01'
           'SOURCE.ENC.01' )

>>> ENCDATA <<< 2007-08-26-12.42.42.5520
Key store designated
SAPI returned 0/0

Hash completed successfully.
SAPI returned 0/0

Signature generation was successful
Signature has length = 64
SAPI returned 0/0

ENC Public Key extracted from key store
SAPI returned 0/0

Random DES-key generated successfully
SAPI returned 0/0

Signature enciphered successfully
SAPI returned 0/0

Random DES-key generated successfully
SAPI returned 0/0

DATA enciphering completed successfully.
SAPI returned 0/0

Press ENTER to end terminal session.

```

- c. Display ciphered data text01.enc and text01.sig:

```

DSPF STMF('/home/MilanK/Files/text01.enc')
*****Beginning of data*****

```



```
5678901234
6789012345
7890123456
8901234567
*****
* text01 *
*****
*****End of Data*****
```

12. Log off from our Cryptographic Coprocessor:

```
CALL PGM(LOGOFF)
Log off successful
Press ENTER to end terminal session.
```

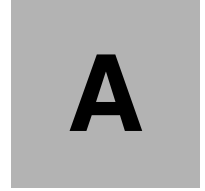
13. Deallocate a Cryptographic Coprocessor device description from our job:

```
CALL PGM(CRPDEALLOC) PARM(CRP02)
Request was successful
Press ENTER to end terminal session.
```

**Note:** A description of SAPI can be found in the manual *IBM PCI Cryptographic Coprocessor, CCA Basic Services Reference and Guide, Release 2.52, IBM iSeries PCICC Feature*, available at:

[http://www-306.ibm.com/security/cryptocards/pdfs/IBM\\_4758\\_Basic\\_Services\\_Release\\_2\\_52.pdf](http://www-306.ibm.com/security/cryptocards/pdfs/IBM_4758_Basic_Services_Release_2_52.pdf).





## Additional material

This book refers to additional material that can be downloaded from the Internet as described below.

### Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

<ftp://www.redbooks.ibm.com/redbooks/SG247399>

Alternatively, you can go to the IBM Redbooks Web site at:

[ibm.com/redbooks](http://ibm.com/redbooks)

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247399.

### Using the Web material

The additional Web material that accompanies this book includes the following files:

<i>File name</i>	<i>Description</i>
<b>SG247399.zip</b>	Zipped code samples for scenarios described in Chapter 11, “Cryptographic Services APIs method” on page 133, and Chapter 12, “HW-based method” on page 229

### System requirements for downloading the Web material

We recommend the following system configuration:

**Operating system** IBM i5/OS V5R4

## **How to use the Web material**

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.



# Related publications

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

## IBM Redbooks

For information about ordering these publications, see “How to get Redbooks” on page 283. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *IBM System i Security Guide for IBM i5/OS Version 5 Release 4*, SG24-6668
- ▶ *IBM eServer iSeries Wired Network Security: OS/400 V5R1 DCM and Cryptographic Enhancements*, SG24-6168
- ▶ *Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries*, SG24-6503

## Other publications

These publications are also relevant as further information sources:

- ▶ *iSeries Security Reference Version 5*, SC41-5302-06
- ▶ *z/OS ICSF Application Programmer's Guide*, SA22-7522

## Online resources

These Web sites are also relevant as further information sources:

- ▶ IBM Information Center, Version 5 Release 4 Web site  
<http://publib.boulder.ibm.com/infocenter/iseriess/v5r4/topic/rzahg/rzahgicsecurity.htm>
- ▶ FIPS Pub 46-3, Data Encryption Standard (DES) description  
<http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>
- ▶ CCA API set description  
<http://www-03.ibm.com/security/cryptocards/library.shtml>

## How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

[ibm.com/redbooks](http://ibm.com/redbooks)

## Help from IBM

IBM Support and downloads

[ibm.com/support](https://ibm.com/support)

IBM Global Services

[ibm.com/services](https://ibm.com/services)

# Index

## Symbols

\*BEFORE 99  
\*CHANGE 99  
\*INSERT 99  
\*JOBCTL 106  
\*SPLCTL 106

## Numerics

2058 22  
2TDES 14  
3TDES 14  
4758 22, 61  
4764 22, 61  
65535 79

## A

access points 3  
accountability 3  
ADDLIBLE 241  
ADDPFTRG 99  
administrative function 7  
Advanced Encryption Standard 11, 14  
AES 11, 14, 48  
algorithms 8  
alter 4  
Alternate character set 78  
analyzing impact to performance 117  
assessments 5  
asymmetric algorithms 16  
at-rest 6  
attorney 45  
auditing 8  
Auditing Infrastructure 69  
auditor 45  
authenticating 4  
authentication 8  
Authentication APIs 133  
auxiliary file 100

## B

backup copy 90  
BER 54  
BINARY 116  
BLOB 116  
block cipher 13  
breach 46

## C

calculate HASH API 181  
Cardholder Information Security Protection 42  
CBC mode 74  
CCA 23, 229

CCA master keys 63  
CCSID 79, 111  
CFB 16, 74  
CHAR 118  
character analysis 100  
CHGOBJOWN 66, 68  
CHGPF 80  
CHGPGM 68  
CHGSECAUD 69  
chip 23  
cipher 12  
Cipher Feedback 16  
ciphertext 7, 12, 100  
CISP 42  
clear key 53  
Clear key parts 62  
clear key values 28  
Clear keys 62  
Clear Master Key Version 144  
clock 56  
COBOL 78  
Common Cryptographic Architecture 23, 229  
compliance 40  
compliance regulations 43  
comply 3  
compromised 33  
confidentiality 4  
context 54  
conversion program 86  
conversions 22  
coprocessor 57  
correcting data sequence 105  
cost of translation 33  
CPU intensive 27  
CPYF 84  
CREATE TRIGGER statement 99  
CRP01 63  
CRP02 63  
CRTDEVCRP 241  
CRTDUPOBJ 54, 84  
CRTLIB 66  
CRTUSRPRF 66  
cryptanalysts 14  
Cryptographic Accelerator 22  
cryptographic algorithm 12  
Cryptographic Context APIs 133  
Cryptographic Coprocessor 22  
cryptographic measures 4  
cryptographic operations 11  
cryptographic service providers 22  
cryptographic terms 8  
Cryptographic Unit Support Program 15  
cryptography 4  
cryptoperiod 33  
CSP 28

- CSPs 22
- CSUACRA 241
- current 48
- CUSP 15
- CUSP mode 74
- customers 3
- CVV 41

## D

- Data Encryption Algorithm 13
- Data Encryption Standard 13
- Data File Utility 90
- data keys 31
- Data Security Operating Policy 42
- data sorting 103
- data-at-rest 6, 8
- database normalization 85, 98
- database structure changes 84
- Database Triggers 85
- data-in-flight 5
- data-in-motion 5
- DB2 table 116
- DCM 23
- DEA 13
- decrypt the message 13
- decryption 8
- defense 3
- DES 13, 33
- DFU 90
- Diffie-Hellman 17
- Digital Certificate Manager 23
- digital media 29
- digital signature 21
- Digital Signature Standard 18
- digital signatures 4
- directives 44
- disaster recovery strategy 5
- DISC 42
- Discover Information Security and Compliance 42
- Display Physical File Member 95
- DSOP 42
- DSPAUDJRNE 69
- DSPPFM 95
- DSPPGMREF 78
- DSPSECAUD 69
- DSS 18, 41

## E

- ECB 15
- ECC 28
- EID 56
- Electronic Code Book 15
- Eliptic Curve Cryptography 28
- employees 3
- EMV 23
- Encrypted external keys 62
- Encrypted internal keys 62
- Encrypted key pairs 62
- encrypted message 13

- encryption 3
- Encryption and Decryption APIs 133
- encryption technology 3
- enterprise 3
- entropy 28
- environment ID 56
- exclusive-ORed 31
- exhaustive search 28
- existing configuration 6
- exit-point technology 8
- exposed tape media 6
- exposure 28
- External key token 58
- External Triggers 99
- Extraction process 89

## F

- Federal Information Processing Standard 13
- Federal Trade Commission 41
- field data length 102
- field data tagging 102
- field level encryption 20
- financial PINs 19
- FIPS 13
- fixed-function program 106
- forward-thinking organizations 5
- frequency 33
- FTC 41
- FTP 8, 240

## G

- GETHINT 128
- GLBA 40
- government 40
- Gramm-Leach-Bliley Act 40
- GRAPHIC 118
- GRTOBJAUT 67

## H

- hardware security module 22
- hash MAC 21
- hash operation 20
- Healthcare Insurance Portability and Accountability Act 41
- HEX 79
- HIPAA 41
- HLL 78
- HMAC 21
- HMAC keys 51
- HSM 22, 57

## I

- i5/OS Cryptographic Services 23
- iASP 111
- impact 46
- Increased data length 78
- industry policy 5
- information assets 3

- infrastructure 5, 40
- initialization vector 15
- instill 44
- integrated circuit 23
- integrity 4
- interactive SQL 94
- intercepted electronic transmission 4
- Internal key token 58
- IPSec 5
- IV 15, 139

## J

- J2SDK 22
- Java 2 Software Development Kit 22
- Java Cryptography Extension 22
- JCE 22
- JLFS 82
- Job Control 106

## K

- KEK 35, 111
- KEKs 31, 51
- key context 54
- Key Description 53
- key destruction 35
- key distribution 22
- Key Generation APIs 133
- Key identifier 60
- key in keystore 53
- key management 22
- Key Management APIs 133
- key size 26
- key token 58
- key values 26
- key verification value 32
- Key version field 82
- Key\_Token\_Parse 58
- keyed hashes 4
- key-encrypting keys 31
- keystore file 51
- KVV 32, 49–50

## L

- larger key size 26
- layers 6
- legislation 39
- legislative requirements 3
- legitimate business purpose 7
- legitimate trade off 82
- length of each passphrase 146
- LIC 18, 30
- Licensed Internal Code 18
- life-span of a key 33
- link level encryption 20
- Load Master Key Part 49
- lock down 3
- Logical file information 85

## M

- MAC 11
- MAC operation 20
- market advantage 5
- master key 30
- Master Key ID 144
- master key parts 32
- MasterCard Site Data Protection 42
- mathematical discipline 4
- mathematical procedure 12
- MD5 17
- MDC 18
- mechanism 81
- media 5
- message authentication 4
- message authentication code 11, 20
- message digest 20
- Message Digest 5 17
- modes of operation 15
- Modification Detection Code 18
- monitor 8

## N

- National Institute of Standards and Technology 14
- needs analysis 40
- NERC 43
- new KEK 65
- new version 49
- NIST 14
- non-compliance 44
- non-digital media 29
- non-SQL 115
- normalization 81
- normalizing the encrypted fields 81
- Null key token 58

## O

- Object-level auditing 69
- object-level controls 8
- ODBC 8
- OFB 16, 74
- old KEK 65
- old master key 48
- one-way hash 17
- organization 39
- Output Feedback 16

## P

- PAN 41
- Parity tagging 103
- partners 3
- Passphrase Part 144
- password 4
- Payment Card Industry 41
- PCI 41
- PEM 51, 53
- performance 26
- perimeter 3

- permanent DFU programs 92
- personal credit information 6
- personal identification number 21
- phased conversion 87
- PHI 41
- physical control 6
- physical file 99
- PIN 21
- PKA 16, 55
- PKA modulus 27
- PKCS 29
- plaintext 12, 14, 46
- Pre-exclusive-OR 61
- Primary Account Number 41
- privacy 3
- Privacy Enhanced Mail 51
- private data 4
- private key 16
- private/public key pair 16
- PRNG 18
- proactive 46
- processes 44
- profile switching 8
- proprietary 5
- protect 3
- Protected Health Information 41
- protected key 47
- protection 8
- pseudorandom number generator 18
- Pseudorandom Number Generator APIs 133
- Public and Private authorities 85
- public domain 12
- public key 16
- public key algorithm 16
- Public-Key Cryptography Standard 29

## Q

- QA 46
- QAC6KEYST 59
- QAC6PKEYST 59
- QAUDCTL 69
- Qc3CalculateHash 149, 178, 181, 190
- QC3CALHA 149, 178, 181, 190
- Qc3CreateAlgorithmContext 161
- Qc3CreateKeyContext 160
- Qc3CreateKeyStore 155, 157–158
- QC3CRTAX 161
- QC3CRTKS 155, 157–158
- QC3CRTKX 160
- QC3DECDT 190
- Qc3DecryptData 190
- Qc3DeleteKeyRecord 168
- QC3DESAX 167
- QC3DESKX 167
- Qc3DestroyAlgorithmContext 167
- Qc3DestroyKeyContext 167
- QC3DLTKR 168
- QC3ENCDDT 179
- Qc3EncryptData 179
- Qc3GenKeyRecord 158

- QC3GENKR 158
- Qc3GenPRNs 176
- QC3GENRN 176
- QC3GENSK 162
- Qc3GenSymmetricKey 162
- Qc3RetrieveKeyRecordAtr 173, 189
- QC3RTVKA 173, 189
- Qc3SetMasterKey 148
- QC3SETMK 148
- Qc3TestMasterKey 175
- Qc3TranslateKeyStore 52, 196
- QC3TRNKS 52, 196
- QC3TSTMK 175
- Qc3WriteKeyRecord 164–165
- QC3WRTKR 164–165
- QLGSORT 105
- QLGSORTIO 105
- QTEMP 105
- quality assurance 46
- quality System i security policy 7
- Query/400 93
- QUSEADPAUT 68

## R

- RACE Integrity Primitives Evaluation Message Digest 18
- ramifications 44
- RC2 14, 75
- RC4 15
- Record layout modification 84
- recovery 56
- Redbooks Web site 283
  - Contact us xii
- reference by the starting/ending positions in the record format 98
- reference the ordered fields by length 98
- Referential Constraints 85
- RIPEMD-160 18
- Rivest Cipher 2 14
- Rivest Cipher 4 15
- Rivest, Shamir, Adleman 17
- RPG 78, 102
- RSA 17
- Run Query 94
- Run SQL Statement 94
- RUNQRY 94
- RUNSQLSTM 94
- RVKOBJAUT 67

## S

- safeguarding your key 12
- Sarbanes-Oxley 40
- Sarbox 40
- SAVLIB 32
- SAVOBJ 32
- SB-1386 42
- scramble 12
- SDP 42
- search 26
- second KEK 65

- secrecy 4
- secret key algorithm 12
- secure channel 137
- Secure Electronic Transaction 23
- Secure FTP 137
- Secure Hash Algorithm 1 18
- Secure Hash Algorithm 2 18
- secure physical channel 34
- security foundation 6
- security level 27
- security policy 7
- segregated test environments 8
- sensitivity 33
- session level encryption 20
- SET 23
- Set Master Key 49
- SHA-1 18
- SHA-2 18
- SHA-256 HMAC 75
- size of key 26
- Sort APIs 105
- SOX 40
- Spool Control 106
- SQL 23
- SQL built-ins 23
- SQL Triggers 99
- SSL 4, 15, 35
- SSN 101
- standards 40
- stream cipher 13
- Structured Query Language 23
- Subfile-based applications 104
- suppliers 3
- Switch profile APIs 68
- symmetric algorithms 12
- symmetric data-encryption keys 34
- system assets 3
- System-level auditing 69

## T

- TDES 14, 74
- temporary DFU programs 91
- temporary mapping file 87
- Test Master Key 50
- track 35
- transformation 12
- translate operation 20
- transport 57
- trigger 86, 99
- triggers 44
- Triple DES 14

## U

- UDF 100, 121, 204
- unauthorized user 6
- unintelligible 4
- unscramble 12
- Update process 89
- User Defined Function 100

- User Defined Functions 121
- using a trigger to decrypt 100

## V

- validation 89
- VARBINARY 116
- VARCHAR 118
- VARGRAPHIC 118
- VPNs 5

## W

- Write Key Record API 51

## X

- XORed 13







## IBM System i Security: Protecting i5/OS Data with Encryption







# IBM System i Security: Protecting i5/OS Data with Encryption



**Understand key concepts and terminology of cryptography**

**Properly plan for i5/OS data encryption**

**See implementation scenarios of data encryption**

Regulatory and industry-specific requirements, such as SOX, Visa PCI, HIPAA, and so on, require that sensitive data is stored securely and protected against unauthorized access and modifications. Several requirements state that data must be encrypted.

IBM i5/OS offers several options that allow customers to encrypt data in database tables. However, encryption is not a trivial task. Careful planning is essential for successful implementation of a data-encryption project. In a worst case, if improperly encrypted, you would not be able to retrieve clear text information of encrypted data.

This IBM Redbooks publication will help planners, implementers, and programmers by providing three key pieces of information:

- ▶ Part 1, “Introduction to data encryption” on page 1, introduces key concepts, terminology, algorithms, and key management techniques. Understanding these is important for following the rest of this book.
- ▶ Part 2, “Planning for data encryption” on page 37, provides critical information about planning a data encryption project on i5/OS.
- ▶ Part 3, “Implementation of data encryption” on page 113, provides various implementation scenarios with step-by-step guide.

**INTERNATIONAL  
TECHNICAL  
SUPPORT  
ORGANIZATION**

**BUILDING TECHNICAL  
INFORMATION BASED ON  
PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:  
[ibm.com/redbooks](http://ibm.com/redbooks)**

SG24-7399-00

ISBN 0738485373