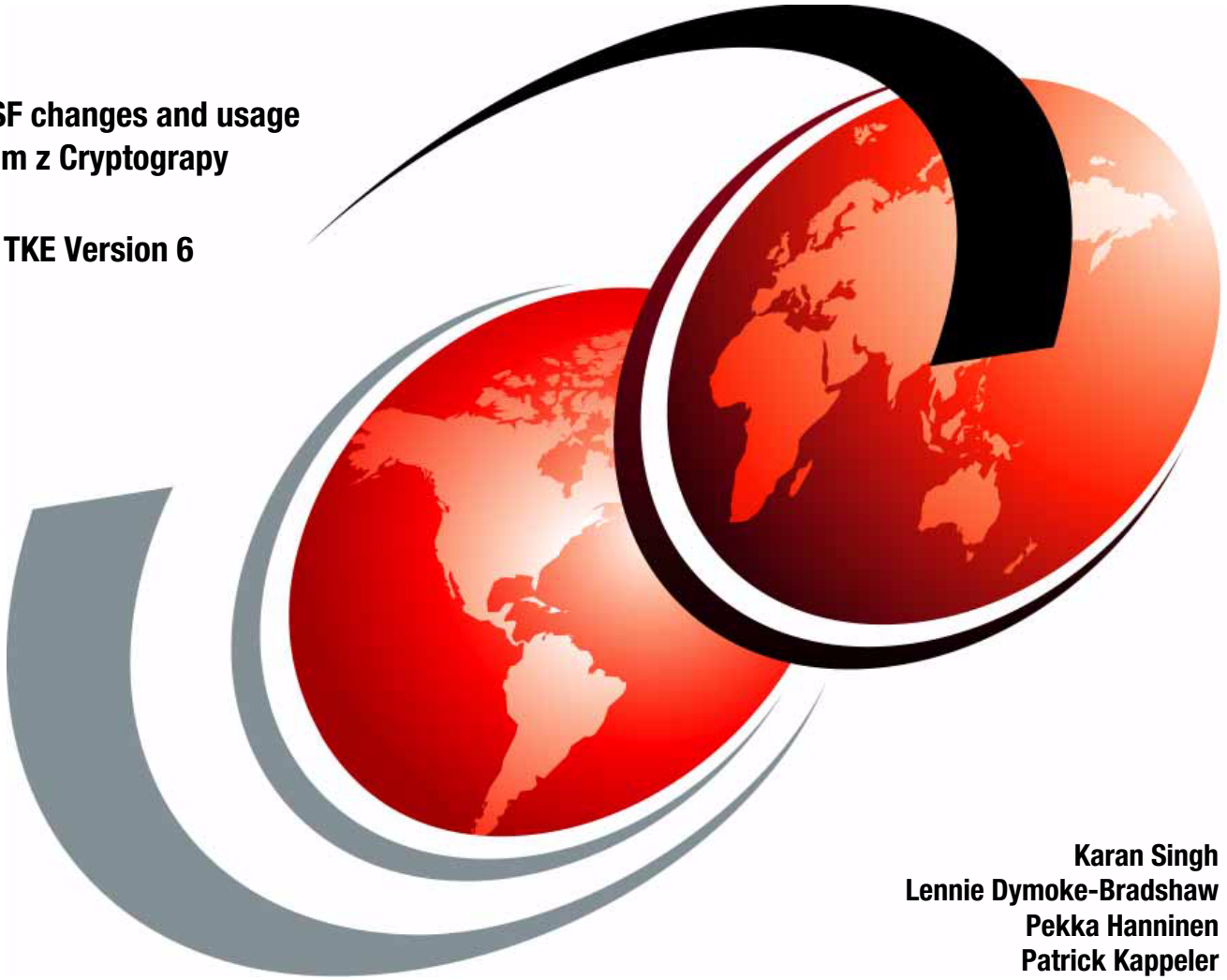
IBM

System z Crypto and TKE Update

- Crypto Express3 technical overview and usage
- Recent ICSF changes and usage with System z Cryptograpy
- Details on TKE Version 6 features



Karan Singh
Lennie Dymoke-Bradshaw
Pekka Hanninen
Patrick Kappeler

Redbooks

ibm.com/redbooks

International Technical Support Organization

System z Crypto and TKE Update

May 2011

SG24-7848-00

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

First Edition (May 2011)

This edition applies to Version 1, Release 12, Modification 0 of z/OS (product number 5694-A01) and TKE Version 6.

Contents

Notices ix
Trademarks x

Preface xi
The team who wrote this book xi
Now you can become a published author, too! xii
Comments welcome. xiii
Stay connected to IBM Redbooks xiii

Chapter 1. Introduction. 1
1.1 Hardware support of cryptographic functions in System z10 2
1.2 First a little bit of history: The IBM mainframe hardware cryptography roadmap 2
1.3 Overview of System z10 hardware cryptographic features. 4
1.3.1 Overview of the CP Assist for Cryptographic Functions (CPACF) 5
1.3.2 The Crypto Express3 feature (CEX3C). 7
1.4 Sharing the Crypto Express3 coprocessor between logical partitions 12
1.5 System z10 and Crypto Express3 configuration information. 12
1.5.1 Maximum number of CEX3C features and coprocessors/accelerators 13
1.5.2 Hardware feature codes referred to in this book. 13
1.5.3 The TKE workstation feature 14
1.6 Cryptographic features comparison 16
1.6.1 Quick summary of the main functional differences between CPACF and CEX3C coprocessor 16
1.6.2 Performance data 18
1.7 Application programming interfaces 18
1.7.1 The z/OS API - ICSF. 18
1.7.2 Hardware cryptography exploitation in z/VSE. 21
1.7.3 Hardware cryptography exploitation in Linux for System z 23
1.7.4 Setting up the hardware cryptography configuration of z/VM 24
1.8 Required software levels. 25
1.8.1 Support for System z10 CP Assist for Cryptographic Function. 25
1.8.2 Crypto Express3 26

Chapter 2. More on the 4765 Cryptographic Coprocessor and the Crypto Express3 feature 27
2.1 IBM hardware cryptographic coprocessors design points. 28
2.1.1 Flexibility 28
2.1.2 Performance 28
2.1.3 Acceleration and security 28
2.1.4 Multiplatform support and interoperability 29
2.1.5 Export control 29
2.1.6 Security certifications 29
2.2 The 4765 Cryptographic Coprocessor 29
2.2.1 Coprocessor principles of operation 30
2.2.2 Hardware overview 31
2.2.3 Coprocessor software layout. 34
2.2.4 Access control for 4765 card functions 35
2.2.5 The CCA cryptographic application program 35
2.2.6 Multiplatform support 36

2.2.7 CCA platforms interoperability 36
2.2.8 Functional differences with the 4764-001 37
2.3 The System z Crypto Express3 feature 37
2.3.1 Export control 37
2.3.2 Specific usage characteristics of the 4765 coprocessor in System z 37
2.4 CCA services provided by the Crypto Express3 39
2.4.1 System z host system CCA software support 39
2.4.2 Cryptographic standards supported by System z CCA implementations 40
2.4.3 The CCA functions performed by the CEX3C coprocessors 40
2.5 The Crypto Express3 TKE interface 41
2.6 Introduction to the Crypto Express3 User Defined Extension (UDX) support 42
2.6.1 The UDX development process 43
2.6.2 UDX Installation 44
2.6.3 The UDX functional implementation 44
2.6.4 Details on the UDX implementation 45
2.7 Crypto Express3 performance 46
2.7.1 Symmetric encryption performance: Crypto Express3 versus CPACF and Crypto Express2 47
2.7.2 Asymmetric encryption performance: Crypto Express3 versus Crypto Express2 in coprocessor mode 48
2.7.3 Asymmetric encryption performance: Crypto Express3 versus Crypto Express2 in accelerator mode 49

Chapter 3. CP Assist for Cryptographic Functions (CPACF) 51
3.1 Introduction to CPACF 52
3.1.1 Enablement 52
3.1.2 What the Message-Security Assist instructions do 53
3.2 Calling the CPACF via ICSF services 55
3.3 History of CPACF 55
3.4 Protected key theory 59
3.5 How to use CPACF protected keys 61
3.5.1 Method 1: Using ICSF secure keys as protected keys 62
3.5.2 Method 2: Using ICSF clear keys as protected keys 67
3.5.3 Comparison of methods 69
3.6 More details about CPACF protected keys 70
3.6.1 Types of keys and encryption 70
3.6.2 Verification patterns 70
3.6.3 CPACF instructions for protected keys 71
3.6.4 ICSF APIs for protected keys 71
3.6.5 TLB in HSA 71
3.6.6 Performance 71
3.6.7 Key storing 72
3.7 Exploiters of CPACF protected keys 72
3.8 Summary 72

Chapter 4. z10 Cryptographic configuration 73
4.1 Usage domain zeroize 74
4.2 Viewing and changing LPAR Cryptographic Controls 76
4.2.1 View LPAR Cryptographic Controls 77
4.2.2 Dynamically changing LPAR Cryptographic Controls 78
4.3 LPAR Cryptographic and security definitions 79
4.3.1 LPAR Cryptographic definitions 79
4.3.2 LPAR security settings 80

Chapter 5. Integrated Cryptographic Service Facility (ICSF): Recent changes 83
5.1 Introduction 84
5.2 ICSF levels and capabilities 84
5.3 Installation issues 85
5.4 Modes of operation 85
5.4.1 ICSF with CEXnC 85
5.4.2 ICSF with CPACF 86
5.4.3 ICSF without CPACF 87
5.5 Key store policies 87
5.5.1 The need for a a key store policy 88
5.5.2 Key store policy checking 92
5.5.3 Token checking 92
5.5.4 Default token checking 93
5.5.5 Preventing duplicate tokens 94
5.5.6 Key label access levels 96
5.5.7 Symmetric key export controls 98
5.5.8 PKA key management extensions control: Part 1 99
5.5.9 PKA key management extensions control: Part 2 101
5.5.10 Messages 102
5.6 RACF segment for ICSF 102
5.6.1 ASYMUSAGE 103
5.6.2 SYMEXPORTABLE 103
5.6.3 SYMEXPORTCERTS 103
5.6.4 SYMEXPORTKEYS 103
5.6.5 SYMCPACFWRAP 103
5.7 RACF resources for ICSF 103
5.7.1 CSFSERV 104
5.7.2 CSFKEYS 104
5.7.3 DATASET 104
5.7.4 XFACILIT 105
5.7.5 XCSFKEYS 105
5.7.6 CRYPTOZ 105
5.8 Secure AES keys 105
5.9 RSA key lengths 107
5.10 Key stores in a sysplex 108
5.10.1 Background information about the PKDS and TKDS 109
5.10.2 How to enable Sysplex sharing 109
5.10.3 Restrictions on PKDS sharing 110
5.10.4 Changing the PKA master key when the PKDS is shared in a sysplex 110
5.10.5 PKDSCACHE 111
5.10.6 CSFPUTIL 111
5.11 PAN lengths 111
5.12 Update on ICSF APIs since HRF7730 111
5.12.1 ICSF FMID HCR7770 112
5.12.2 ICSF FMID HCR7751 112
5.12.3 ICSF FMID HCR7750 113
5.12.4 ICSF FMID HCR7740 114
5.12.5 ICSF,FMID HCR7731 114
5.13 Auditing records 115
5.13.1 Type 82 (ICSF record) 115
5.13.2 Type 70 - Subtype 2 (RMF Processor Activity) 116
5.13.3 Type 30 (Common Address Space Work) 116
5.13.4 Type 72 - Subtype 3 (Workload Activity) 117

5.14 64-bit services 117
5.15 ICSF query services 118
5.16 The z/OS Health Checker and ICSF 119
5.17 UDX. 120

Chapter 6. z/OS support for the PKCS#11 API 121
6.1 Public Key Cryptography Standard #11 (PKCS#11) 123
6.1.1 An overview of the PKCS#11 concepts 123
6.1.2 The benefits of implementing PKCS#11 on z/OS 124
6.1.3 Mapping PKCS#11 concepts to z/OS cryptographic services implementation . . 124
6.2 z/OS PKCS#11 infrastructure and setup. 125
6.2.1 Tokens 126
6.2.2 The token key data set 126
6.2.3 Controlling access to tokens and objects 128
6.3 z/OS PKCS#11 services 130
6.3.1 The z/OS PKCS#11 C API 130
6.3.2 Supported algorithms 131
6.3.3 Elliptic Curve Cryptography support 132
6.4 Basics of token administration using ICSF token browser 133
6.5 Exploiting the PKCS#11 services 139
6.5.1 Making the services available 139
6.5.2 Learning with the tespkcs11 program 139
6.5.3 Using the ICSF PKCS#11 callable services 147
6.6 Operating in compliance with FIPS 140-2. 153
6.6.1 Preparing to comply with FIPS 140-2 154
6.6.2 Verification of the ICSF load module digital signature 154
6.6.3 Directing ICSF to provide FIPS 140-2 compliant services 155
6.7 More on the omnipresent token 158
6.8 More on interactive token administration 158
6.8.1 Examples of RACF token and certificate management options 158
6.8.2 Examples of gskkyman token and certificate management options 166
6.8.3 Moving the certificate from PKCS#11 token to the RACF database. 171

Chapter 7. Trusted Key Entry 173
7.1 Introduction to the TKE 174
7.1.1 What can be done with a TKE 174
7.1.2 Why TKE is necessary 175
7.1.3 Smart card support 175
7.1.4 Multiple TKEs 176
7.1.5 Roles and profiles 177
7.1.6 Group logon 178
7.1.7 How TKE communicates with System z, Crypto Express, z/OS, and ICSF 179
7.2 HIstory of TKE. 180
7.2.1 TKE V1/V2 (1997 to 2000) 180
7.2.2 TKE V3 (2000) 180
7.2.3 TKE V4 (2004) 181
7.2.4 TKE V5 (2005) 181
7.2.5 Table of TKE levels and capabilities 181
7.3 The latest TKE, Version 6.0 182
7.3.1 Hardware 182
7.3.2 Domain grouping. 183
7.3.3 Support for saving and restoring host configuration data 193
7.3.4 Admin and auditor tasks 208

7.3.5 TKE application changes 218
7.3.6 Other items 223
7.4 Migration from previous releases 227

Chapter 8. A brief overview of integrated hardware cryptography implementation in Linux for System z 229
8.1 A reminder about Linux for System z cryptographic features 231
8.1.1 The cryptographic options available with Linux for System z 231
8.1.2 Clear keys acceleration support libraries and exploiters 233
8.2 The IBM CCA support library and its exploiters 234
8.2.1 The support infrastructure 235
8.2.2 The services provided by the Crypto Express3 coprocessor 236
8.2.3 Access control for the CCA services 238
8.3 Supported configurations 238
8.4 Management of the CCA application's secure keys 238
8.4.1 Key storage initialization 239
8.4.2 Key tokens management 239
8.4.3 Key storage re-encipher 239
8.4.4 Host-side key caching 239
8.5 Management of the Crypto Express3 coprocessor 240
8.5.1 Using the TKE workstation 240
8.5.2 Using the Command Line Interface panel 241
8.6 Specifics of CPACF support 242
8.6.1 Environment variables that affect CPACF usage 242
8.6.2 Access control points that affect CPACF protected key operations 242
8.6.3 Overview of the CPACF protected key operations in Linux for System z 243
8.7 Performance data 243
8.8 Download sites and Linux distributions CCA support 243

Chapter 9. Using Elliptic Curve Cryptography (ECC) on z/OS 245
9.1 Elliptic Curve Cryptography in a nutshell 246
9.1.1 Another asymmetric algorithm 246
9.1.2 The advantages of Elliptic Curve Cryptography 246
9.1.3 Mathematical background 247
9.1.4 ECC in practice 248
9.2 Our sample program 251
9.2.1 Token creation 251
9.2.2 ECC key pair generation 252
9.2.3 ECC signature generation 257
9.2.4 ECC signature verification 257
9.2.5 Token deletion 259

Appendix A. Sample programs in assembler and REXX 261
REXPK001 REXX program 262
CPACF040 assembler program 266
CPACF050 assembler program 275
CPACF001 JCL 281
CPACF010 assembler program 282
REXCP050 REXX program 287
REXIQF program 291
REXIQA program 299

Related publications 301
IBM Redbooks 301

Other publications 301
How to get Redbooks 301
Help from IBM 301

Index 303

Notices

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

Trademarks

IBM, the IBM logo, and ibm.com 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®
DB2®
eServer™
IBM®
IMS™
Language Environment®
MVS™
Power Systems™
PowerPC®
RACF®
Redbooks®
Redbooks (logo)  ®
RMF™
S/390®
System i®
System x®
System z10®
System z9®
System z®
Tivoli®
VTAM®
WebSphere®
z/Architecture®
z/OS®
z/VM®
z/VSE™
z10™
z9®
zSeries®

The following terms are trademarks of other companies:

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

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

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

This IBM® Redbooks® publication provides detailed information about the implementation of hardware cryptography in the System z10® server. We begin by summarizing the history of hardware cryptography on IBM Mainframe servers, introducing the cryptographic support available on the IBM System z10, introducing the Crypto Express3 feature, briefly comparing the functions provided by the hardware and software, and providing a high-level overview of the application programming interfaces available for invoking cryptographic support.

This book then provides detailed information about the Crypto Express3 feature, discussing at length its physical design, its function and usage details, the services that it provides, and the API exposed to the programmer. This book also provides significant coverage of the CP Assist for Cryptographic Functions (CPACF). Details on the history and purpose of the CPACF are provided, along with an overview of cryptographic keys and CPACF usage details. A chapter on the configuration of the hardware cryptographic features is provided, which covers topics such as zeroizing domains and security settings. We examine the software support for the cryptographic functions available on the System z10 server. We look at the recent changes in the Integrated Cryptographic Service Facility (ICSF) introduced with level HCR7770 for the z/OS® operating system. A discussion of PKCS#11 support presents an overview of the standard and provides details on configuration and exploitation of PKCS#11 services available on the z/OS operating system.

The Trusted Key Entry (TKE) Version 6.0 workstation updates are examined in detail and examples are presented on the configuration, usage, and exploitation of the new features. We discuss the cryptographic support available for Linux® on System z®, with a focus on the services available through the IBM Common Cryptographic Architecture (CCA) API. We also provide an overview on Elliptical Curve Cryptography (ECC), along with examples of exploiting ECC using ICSF PKCS#11 services. Sample Rexx and Assembler code is provided that demonstrate the capabilities of CPACF protected keys.

The team who wrote this book

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

Karan Singh is a Project Leader at the International Technical Support Organization, Poughkeepsie Center.

Lennie Dymoke-Bradshaw is a long-term MVS™ practitioner. He has worked with S/370 since 1975, when he started programming a 370/145 running DOS/VS in PL/1 and assembler. Over the years he has maintained his interest in programming while working in a variety of roles in applications programming, systems design, technical support, systems programming, software support, and any other technical roles customers have asked of him. During this time he has worked with every level of RACF® since RACF 1.3, and he maintains a keen interest in computer security, especially as applied to System z. Lennie joined IBM in 1996, and now he works more with System z cryptography. Lennie currently works in IBM System z software pre-sales in the UK, and works to publicise the enormous benefits of System z from a security standpoint.

Pekka Hanninen is a working Crypto and Security Specialist with D7C8 Consulting company based in Helsinki, Finland. He was an IT Specialist for over 12 years in IBM Finland until his

recent retirement. He has over 35 years of experience in IBM large system software. His areas of expertise include cryptography, RACF, and security administration. He holds certificates for CISSP, CISA, and CISM.

Patrick Kappeler is a former IBM technical expert in the domain of IBM System z eBusiness Security. He practiced for the last decade both as an IBM Certified Consulting IT Specialist in the Montpellier (France) European Products and Solutions Support Center (PSSC) and as an IBM Redbooks publication co-author and Project Leader for the International Technical Support Organization (ITSO) in the IBM Poughkeepsie (USA) Laboratory. Patrick now provides worldwide independent consultancy and education on IBM mainframes Security, covering areas such as System z hardware integrated cryptography, RACF, Public Key Infrastructure, secure protocols, IP Security, and LDAP.

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

David Benin, Rich Conway, Bob Haimowitz
International Technical Support Organization, Poughkeepsie Center

Todd Arnold, Wai Choi, Chuck Gainey, Jack Hoarau, Michael Jordan, Kenneth Kerr, Robert Petti, Peter Spera, James Sweeny, Gary Sullivan
IBM

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author - all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

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

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

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

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

1

Introduction

System z10 GA3 implements enhanced functions of the cryptographic facilities already available in System z10 GA1 and GA2. These are the CP Assist for Cryptographic Functions (CPACF) and the Crypto Express feature, which comes with the Crypto Express3 model, replacing the previously available Crypto Express2 feature. This chapter is to remind readers of the hardware cryptography infrastructure in System z, introduce the cryptographic services that it provides in System z10 GA3, and provide information about their relevant exploitation environment.

Note that former implementations of these facilities are described in IBM Redbooks publications *IBM eServer zSeries 990 (z990) Cryptography Implementation*, SG24-7070, and *z9®-109 Crypto and TKE V5 Update*, SG24-7123.

1.1 Hardware support of cryptographic functions in System z10

System z10 includes both standard cryptographic hardware and optional cryptographic features to meet today's installations requirements for flexibility and growth capability. IBM is still capitalizing on its long history of providing hardware cryptographic solutions, from the development of Data Encryption Standard (DES) in the 1970s to delivering integrated cryptographic hardware in a server to achieve the U.S. government's highest FIPS 140-1 Level 4 rating for secure cryptographic hardware in 1998.

FIPS 140-2 is today's prevailing standard when it comes to certifying commercial cryptographic module's implementations and operations. Information about the FIPS 140-2 standard can be found at:

http://csrc.nist.gov/cryptval/140-2.htm

The hardware cryptographic services provided in System z10 are intended to cover the full range of cryptographic operations that are needed for e-business, e-commerce, and financial institution applications, both from the functional and performance standpoints.

The capability for users to design their own customized cryptographic functions to be performed by System z hardware (User Defined Extensions, UDX) is carried over from previous implementations onto System z10 GA3.

This chapter provides an overview of the cryptographic hardware features that are available on System z10 from the functional and configuration standpoints. We also provide a high-level description of the API provided in IBM System z operating systems for products or users' programs to invoke the hardware cryptographic functions.

1.2 First a little bit of history: The IBM mainframe hardware cryptography roadmap

Providing historical data helps us understand the current design philosophy and roadmap for System z hardware cryptography technology implementation. Figure 1-1 on page 4 is a graphical representation of the evolution of these implementations on IBM mainframes for the past decade (beginning with the CMOS mainframe implementation):

- The Cryptographic Coprocessor Facility (CCF) was integrated into the system as two identical functional units (for throughput and availability reasons). The CCF provided the legacy functions that were available on preceding IBM cryptographic devices, mainly oriented towards meeting the banking institutions needs for symmetric algorithm-based cryptographic services, and also introduced the set of asymmetric algorithms-based functions that the rising e-business industry was requesting.

  Note: It is worth noting here that IBM took the position, since the late 1980s, of providing hardware cryptography capabilities to platforms other than mainframes, including small systems and distributed infrastructures. This led to the development of a set of common hardware products, in different versions, and a common cryptographic API: the IBM Common Cryptographic Architecture (CCA) API. In regards to this, the CMOS CCF was very much a mainframe-specific implementation, but was implementing the CCA API.

  The history of IBM cryptographic hardware devices and its influence on the design of current products is also discussed in Chapter 2, "More on the 4765 Cryptographic Coprocessor and the Crypto Express3 feature" on page 27.

- About the year 2000, a new set of cryptographic hardware coprocessors was available for the CMOS mainframes. It complemented the asymmetric algorithms-based functions already available in CCF and provided functional compatibility with CCA-compliant applications developed on other IBM platforms. These were an offspring of the development effort for hardware cryptographic devices that can be exploited on different platforms and came as Peripheral Component Interconnect (PCI) pluggable cards. These cards are inserted in mother boards located in the mainframe I/O cage. These were:
  - The PCI Cryptographic Coprocessor (PCICC), which was also supporting, among other things, the functions available on the IBM 4753 Network Security Processor and the User Defined Extensions capability. (The UDX implementation and process is further discussed in Chapter 2, “More on the 4765 Cryptographic Coprocessor and the Crypto Express3 feature” on page 27.)
  - The PCI Cryptographic Accelerator (PCICA), which aimed at providing maximum throughput for the RSA-based computations required by the SSL/TLS protocol handshake. The PCICA design was deliberately favoring speed at the cost of lowering the security level as the PCICA was using clear text RSA private keys only.
- The System z990 partially changed the implementation model by removing the CCF and introducing the CP Assist for Cryptographic Functions (CPACF) facility. Note that the CPACF was not a replacement for the CCF, but came with its own set of functions. The CPACF is still implemented, with improvements, in the System z10 GA3, keeping the same overall characteristics:
  - There is a dedicated CPACF in each processing unit (PU) of the system. CPACF is not dedicated to each PU on a z10, only on previous modes.
  - The CPACF supports symmetric and one-way hash algorithms only, at very high speed.
  - Secret keys required by the CPACF are clear text keys. That is, their secret value appears in “clear”, not encrypted, within a program.

  However, part of the z990 implementation of hardware cryptographic functions still relied on the PCI card technology, which improved on the z990 with the PCI Extended Cryptographic Coprocessor (PCIXCC) feature, which eventually was replaced by the Crypto Express2 (CEX2C) feature (PCIXCC and CEX2C being functionally equivalent, as the CEX2C was just packaging two PCIXCC coprocessors in the same System z990 feature).

  Finally, in the System z990 implementation philosophy, the PCIXCC/CEX2C coprocessor was picking up all the functions provided by the CCF, PCICC, and PCICA on previous systems (the CEX2C could also perform in accelerator mode, but not on a z990/z890) and was providing new unique functions with the CPACF, to the extent that these functions could run with clear text keys only. A unique feature of CPACF is that it can be called by any program using a publicly available set of instructions.

Figure 1-1 shows the IBM mainframe hardware cryptography roadmap.

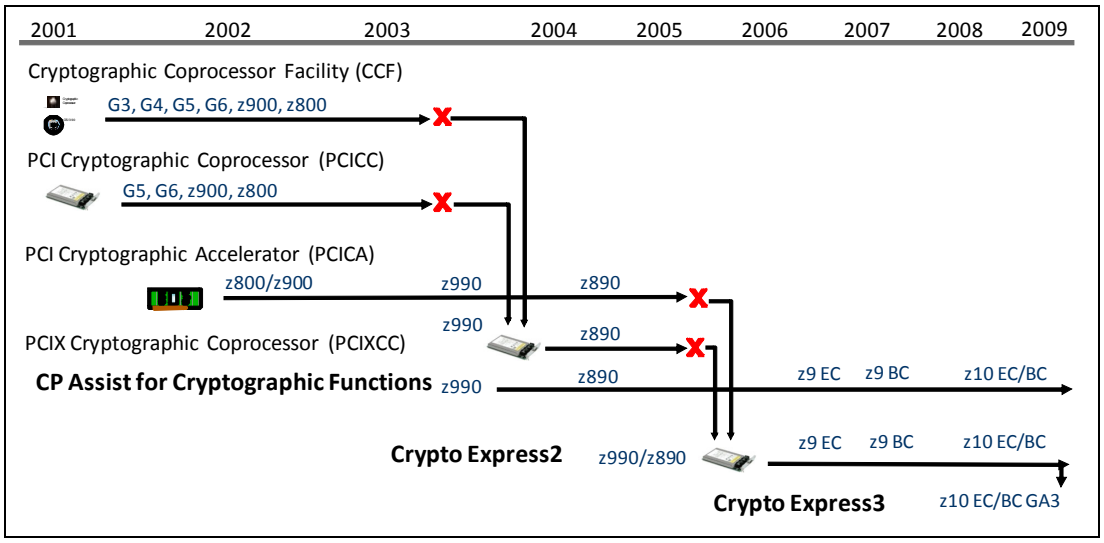


Figure 1-1 IBM mainframe hardware cryptography roadmap

- The System z10, as explained below, inherits an improved version of the technologies implemented in the Systems z990, the CPACF, and the Crypto Express feature. System z10 GA3 introduces the protected key option for CPACF operations, where the secret key is no longer provided in clear text form but is encrypted under a *wrapping key*, and the Crypto Express3 (CEX3C) feature, which supersedes the Crypto Express2.

Note: Although CEX2C and CEX3C features can be mixed in the same System z10, this book focuses on the CEX3C feature only.

1.3 Overview of System z10 hardware cryptographic features

Two types of hardware cryptographic features are available to be installed in a System z10:

- The CP Assist for Cryptographic Functions (CPACF): This is a base feature implemented as a functional extension to the System z10 PU. The CPACF executes at machine speed the Message Security Assist (MSA) z/Architecture® instructions. As explained below, the Feature Code 3863 has to be installed in the machine for the CPACF to provide its full range of cryptographic functions.
- The Crypto Express3 feature (CEX3C): This is an optional priced feature that comes as a “book” to be plugged in the system’s I/O cage. Up to eight of these books can be plugged for cryptographic throughput scalability purposes in the system. The CEX3C feature requires FC 3863 to be installed to provide any cryptographic functions. Note that the acronym CEX3C stands for Crypto Express3 Coprocessor. We explain later the subtle difference between *feature* and *coprocessor* and how a cryptographic accelerator (CEX3A) compares to a coprocessor.

An API is provided in the IBM System z operating systems for invoking these hardware facilities and providing full cryptographic operations support (although, as mentioned above, programs can elect to directly invoke the CPACF using the MSA instructions set).

For z/OS this API is implemented in the Integrated Cryptographic Service Facility (ICSF) component of z/OS. ICSF and its API is briefly described in 1.7.1, “The z/OS API - ICSF” on

page 18, and described in more detail in Chapter 5, “Integrated Cryptographic Service Facility (ICSF): Recent changes” on page 83.

The z/VSE™, z/TPF, and Linux operating systems also provide a APIs, which are briefly described later in this chapter. Note that further details on the Linux for System z hardware cryptography implementation are provided in Chapter 8, “A brief overview of integrated hardware cryptography implementation in Linux for System z” on page 229.

We also provide a brief description of how hardware cryptographic services are exploited in a z/VM® environment. z/VM is not providing an API per se, but requires specific configuration parameters to be set up so that guest VMs can get access to the crytographic services. This is explained at 1.7.4, “Setting up the hardware cryptography configuration of z/VM” on page 24.

The Trusted Key Entry (TKE) workstation is an optional priced feature that consists of an IBM-specific workstation aimed at providing a highly secure environment for the centralized management of the Crypto Express3 coprocessors, in one or more systems, to which the TKE workstation has TCP/IP connectivity. As of the writing of this book, the TKE workstation is available at Version 6.0. It is further discussed in Chapter 7, “Trusted Key Entry” on page 173.

1.3.1 Overview of the CP Assist for Cryptographic Functions (CPACF)

Each system’s PU comes with this integrated assist facility in support of cryptographic instructions. The CPACF can be considered, from the externals standpoint, as a specialized unit in the PU hardware (CPACF is shared between two PUs on a z10) in charge of providing very high-speed cryptographic computations. This functional area of the PU is invoked by a program by issuing Message Security Assist (MSA) instructions. As of the writing of this book, the z/Architecture offers the following set of Message Security Assist instructions. (Refer to *z/Architecture Principles of Operation*, SA22-7832, for further details.)

- KMAC: Compute message authentication code

  This instruction implements the MAC algorithm used for message integrity and authentication checking using the DES or Triple-DES encryption algorithm.

- KM: Cipher message

  This instruction provides encryption and decryption of data using the DES, Triple-DES, or AES algorithm in Electronic Code Book (ECB) mode.

- KMC: Cipher message with chaining

  The KMC instruction also provides encryption and decryption of data using the DES, Triple-DES, or AES algorithm and implements the Cipher Block Chaining (CBC) function as part of the encryption/decryption process.

  The KMC instruction can also be used to invoke a pseudo-random numbers generator function implemented in the CPACF.

Important: Encryption and decryption of data in the CPACF is performed using a secret key passed by the application. Prior to System z10 G3, this encryption key was provided only in clear text form, meaning that the value of the key appears in the program-addressable memory of the system. It therefore requires the use of proper and strong access control setups to preserve the secrecy of this key value. With System z10 G3, the CPACF can be optionally provided with a *protected key* (that is, a key itself encrypted under another key). The secrecy of this latter key-encrypting-key is ensured by built-in system functions. For more information about the CPACF protected key see Chapter 3, “CP Assist for Cryptographic Functions (CPACF)” on page 51.

- KIMD: Compute Intermediate Message Digest

  The KIMD instruction provides the Secure Hash Algorithm (SHA) function for processing segments of an input string of data.

- KLMD: Compute Last Message Digest

  The KLMD instruction produces a SHA result for a single-segment input data string or for the final segment of a multi-segment input data string.

- PCKMO

  The PCKMO instruction is to be used in a program that is intended to generate and manage CPACF protected keys by itself. This is part of the protected key discussion can be found in 3.4, “Protected key theory” on page 59.

All these instructions, except for the PCKMO instruction, are operating in problem state and can therefore be issued by any program that executes in the system.

Note that ICSF also provides a higher level API for invoking the CPACF functions for those programs where, for various reasons, the sole use of MSA instructions does not fit the requirements. This is discussed further in Chapter 5, “Integrated Cryptographic Service Facility (ICSF): Recent changes” on page 83.

Important: FC 3863 has to be installed in the system for the KMAC, KM, and KMC to operate. Without FC 3863 installed, the CPACF can only provide the SHA function.

Algorithms and key lengths supported by the System z10 CPACF

Table 1-1 is a summary of this information. Chapter 3, “CP Assist for Cryptographic Functions (CPACF)” on page 51, provides more details.

Table 1-1 Functions provided by System z10 G3 CPACF

Function	Algorithm	Effective key length (bits)
Data encryption/decryption	DES T-DES AES	56 (1) 112 or 168 (1) 128, 192 or 256
Message authentication	DES TDES	56 (1) 112 or 168 (1)
Message integrity	SHA-1 SHA-256 SHA-512	N/A

Note (1): These are the effective lengths of DES and Triple DES keys. The key lengths can also be expressed as multiples of 64 bits (a 64-bit block holds a single DES key plus eight program-readable parity bits), therefore yielding in this case key lengths of 64, 128, and 192 bits.

Specific characteristics of the CPACF facility

As a result of the CPACF's functional implementation in the system's PUs, the CPACF exhibits two specific characteristics:

- The cryptographic functions that it provides are said to be synchronous, meaning that the execution of the instruction flow by the invoking PU is held for the time that it takes to execute the MSA instruction.
- The CPACF facility was not submitted, as of the writing of this book, for certification under commonly accepted cryptographic standards.

1.3.2 The Crypto Express3 feature (CEX3C)

The optional Crypto Express3 comes as a Peripheral Component Interconnect Express (PCIe) pluggable priced feature that provides high-performance cryptographic functions and a secure cryptographic environment. This section provides high-level information about the coprocessor operational characteristics. Further details are given in Chapter 2, "More on the 4765 Cryptographic Coprocessor and the Crypto Express3 feature" on page 27.

The meaning of secure environment

The Crypto Express3 feature implements the concept of *secure key*, where the secret cryptographic keys that programs use are themselves encrypted under a so-called *master key*. The master key is a secret key that resides inside the protected physical boundary of the CEX3C coprocessor only. The master key never appears outside the coprocessor physical boundary, as there is no function built into the coprocessor that would allow retrieval of the master key value. Therefore, you must ensure its absolute secrecy and the secrecy of the key that. its encrypts. Figure 1-2 illustrates this.

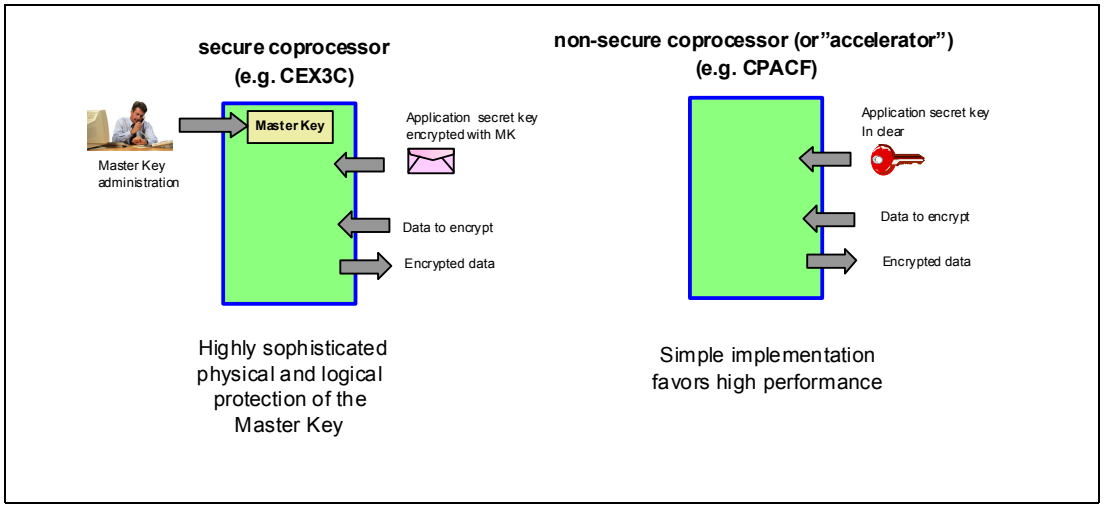


Figure 1-2 The master key concept implementation

The master key conceptual implementation is shown on the left in Figure 1-2, where the applications' secret keys are kept when outside the physical boundary of the secure coprocessor, encrypted under the master key (represented by the envelope here). The application key is retrieved in its clear form by decryption under the master key in the coprocessor and is then used, still inside the coprocessor, to encrypt the string of data passed by the application.

Terminology: The terminology in use with IBM implementations refers to *secure keys* to designate keys encrypted under a master key. These keys are also called *operational keys* when being referred to in their utilization context.

On the contrary, a non-secure coprocessor (also called an *accelerator*) does not host any master key and deals only with the clear text form of the applications' secret key.

It is worth noting here that proper protection of the master key against tampering attempts in a secure coprocessor requires the implementation of sophisticated protection functions based on physical protection and multiple sensor-based circuits. In contrast, each circuit that is implemented on a non-secure coprocessor electronic chip does participate in providing the best possible operating performance of the device.

The CEX3C is a *secure coprocessor* because it hosts a master key, despite the fact that, as seen later, it can be set up to run in accelerator mode. The CPACF remains a non-secure coprocessor, even when using the System z10 GA3 protected key option. This option does not use the master key approach. See Chapter 3, "CP Assist for Cryptographic Functions (CPACF)" on page 51, for a detailed explanation about how the CPACF protected key option works.

CEX3C feature and CEX3C coprocessor

The CEX3C feature hosts one or two coprocessors. As previously mentioned, IBM has developed several versions of a common cryptographic coprocessor technology so that it can be installed and exploited in different families of computers. This common technology is based on the IBM 4765 coprocessor card. The Crypto Express feature is actually the System z10 "book" that hosts one or two 4765 coprocessors:

- The Crypto Express3 feature with Feature Code 0864 hosts two coprocessors and can be installed either in System z10 EC or BC.
- The Crypto Express3-1P feature with Feature Code 0871 hosts only one coprocessor and can be installed in System z10 BC only.

The maximum number of coprocessors available on a single System z10 can therefore be 16 coprocessors when eight Feature Code 0864s are installed.

Note that each coprocessor acts independently from the other coprocessors, therefore increasing the number of coprocessors, potentially increasing the global cryptographic throughput of the system. Increasing the number of coprocessors does not speed up the execution of individual cryptographic services.

Attention: The documentation refers to a *card* to designate a coprocessor, and what is actually plugged into the system's I/O cage is a *feature*.

In this book we use *4765* and *CEX3C coprocessor* interchangeably. *AP* (adjunct processor) is also commonly used to designate a coprocessor.

The functions performed by the CEX3C coprocessor

Contrary to the CPACF, there are no z/Architecture instructions that can be used in customers' programs to invoke the CEX3C services. All requests must be issued through the operating-system-provided API.

The IBM API provided for programs to interact with the 4765 coprocessor is the IBM Common Cryptographic Architecture (CCA) API. The CCA API exposes elementary functions that are

provided by the coprocessor itself or, for some cryptographic services, aggregates coprocessor's functions in a higher level process that composes the specific service.

The high-level families of functions built in the 4765 coprocessor and that are relevant to its use in the System z10 are described below.

Attention: Not all the functions available in the 4765 coprocessor are exploited in its implementation in the System z10 Crypto Express3 feature. Our own experience has been to search for examples of use of some functions in the System z10 context, finally learning that the function was, for example, available on distributed platforms only.

Master keys

The 4765 hosts three master keys dedicated to protecting three families of program-owned cryptographic keys:

- The DES master key (DES-MK) is dedicated to the encryption of DES and Triple-DES secure keys. This is a double length (112 bits) Triple-DES key.
- The AES master key (AES-MK) is used for the encryption of AES secure keys. This is an AES 256-bit long key.
- The PKA master key (ASYM-MK) is used for the encryption of RSA secure private keys and for encryption of the integrity checking data of the ICSF *trusted block* entities. This is a triple length (168 bits) Triple-DES key.

The coprocessor also implements the functions required for a secure management of the Master keys.

Data protection using symmetric algorithms

The functions below are performed using secure keys only in the CEX3C coprocessor:

- Data encryption and decryption
  - Using the DES or Triple-DES algorithm
  - Using the AES algorithm
- DES and AES key generation, importation, and distribution

  The key generation can be random generation or based on a derivation algorithm.

  The symmetric key can be exported in a format compatible with remote devices such as ATMs for an automated remote loading of the key.

Data integrity using symmetric and one-way algorithms

For data integrity:

- Message Authentication Code (MAC), Modification Detection Code (MDC), and one-way hash algorithms are provided.

  The MAC services require the use of secure keys.
- SHA-1 and SHA-2 (SHA-256, SHA-384, and SHA-512) support.

Asymmetric algorithms support

As of the writing of this book, the CEX3C coprocessor provides only RSA hardware support to System z users through the CCA API. There is, however, an internal implementation of the Diffie Hellman algorithm, which is reserved for the coprocessor's own use when securely communicating with the TKE workstation.

Note also that the CCA API refers to Public Key Algorithm (PKA) services to generally designate the RSA services that the CEX3C coprocessor provides.

Depending on the function invoked, the RSA secret private key is required to be presented in secure key form or in clear text form.

As of the writing of this book, the maximum RSA key length supported by the CEX3C feature is 4096 bits.

- Support of the RSA key pairs management
  - Importation of public-private key pairs in clear text and externally encrypted forms
  - Key pair generation
  - RSA key format translation from 4765 format to smart card format.
- RSA Digital signature function

  These services require to use a secure RSA private key.
  - Generation, with selection of the signature formatting method.
  - Verification
- Symmetric key encryption function

  The CEX3C coprocessor can proceed with the encryption of secure symmetric keys (DES, Triple-DES or AES) by an RSA public key, for secure export of these keys, or by an RSA private key when importing such a secure key.

  A specific function is also implemented to assist the symmetric key wrapping or unwrapping during the SSL/TLS protocol handshake with formatting of the resulting symmetric key block.
- Modular exponentiation

  This function is a specific form of the symmetric key encryption function, which programs can invoke whenever such an exponentiation has to be applied against a data string. This can be used, for instance, for assisting software implementation of asymmetric algorithms other than RSA that also rely on modular exponentiation.
- Support of RSA *retained keys*

  Specifying RETAINED is an option when invoking and generating RSA key pair generation by the coprocessor. The key pair is generated in the secure physical boundary of the coprocessor and the private key is never to leave this physical boundary. The retained private key, when needed, is designated by a label and retrieved by the coprocessor from its protected internal memory.

  Note: Although RSA retained keys are supported in the CEX3C coprocessor, their use in System z is not suggested by IBM. The retained keys technologies are intended to be used on other platforms hosting their own version of the IBM 4765 coprocessor.

Financial services support

These services require the use of secure keys:

- Personal identification number (PIN) processing services: Typically generation, verification, and translation, with support for different formats of the PIN block.
- Credit Card verification value or card security code processing: Typically generation and verification, with support for different card types and algorithms.
- Support for secure messaging as implemented in the Europay - Master Card-Visa (EMV) protocol.
- Support for composing/decomposing data blocks in the Secure Electronic Transfer (SET) protocol.

Other functions supported

The Crypto Express3 hardware also implements a high-quality and high-speed random number generator.

Implementation of customized functions in the coprocessor

Users can customize the functions provided by the CEX3C coprocessor by creating a User Defined Extension (UDX) firmware module that performs the user-designed function as a part of the coprocessor firmware. The creation of UDX modules requires the use of a specific toolkit owned by IBM or approved business partners.

Access control for the provided functions

The CEX3C coprocessor exploits the access control points (ACP) mechanism for the purpose of controlling access to functions that it provides. ACPs are logical switches in the coprocessor firmware that enable or disable the use of specific functions. In System z the ACPs can be acted upon via the TKE workstation only.

The certification of the 4765 coprocessor

The 4765 coprocessor's predecessor, the 4764-001 coprocessor, has been certified to meet the FIPS 140-2 standard level 4 compliance criteria. The 4765 has also been designed to meet this standard and level and, as of the writing of this book, it is undergoing the certification process.

The objectives of the FIPS 140-2 standards are discussed in further detail in Chapter 2, "More on the 4765 Cryptographic Coprocessor and the Crypto Express3 feature" on page 27.

Modes of operations of the CEX3C coprocessor

By design, the coprocessor can be set up to run in either one of the following two modes:

- The coprocessor mode: This is the default mode of operations, where the full range of cryptographic functions is available to programs invoking the coprocessor, using secure keys whenever required. When it comes to implicitly stating that the coprocessor is performing in coprocessor mode, the documentation usually refers to a CEX3C.
- The accelerator mode: In this mode of operations the coprocessor provides a very limited support that resolves into performing only the RSA operations that are used during the SSL handshake (these are symmetric key wrapping and unwrapping and digital signature verification). However, these operations can then be performed at a higher speed.

  Secure keys are not supported in accelerator mode (that is, only clear text RSA private keys are used). When it is necessary to implicitly state that the coprocessor is performing in accelerator mode, it is designated as a CEX3A.

Switching between coprocessor and accelerator mode is a manual operation that is performed against a selected coprocessor at the system's Hardware Management Console (HMC) or Service Element (SE). It is possible to mix in the same CEX3C feature a coprocessor running in coprocessor mode and the other one running in accelerator mode.

Synchronous and asynchronous operations

All CEX3C operations are synchronous with respect to the calling program, as the program execution is suspended waiting for the API to return the result of the operation.

However, as seen from the API software, all CEX3C operations are asynchronous. That is, the API software sends a request for operation to the coprocessor and processes other tasks while waiting for the coprocessor to respond.

Running in such an asynchronous mode is something that one can expect when performing asymmetric algorithm operations, as these are very long duration operations compared to the instruction execution speed of a PU.

In the case of the CEX3C coprocessor, despite its intrinsic speed of execution, symmetric algorithms operations are also run asynchronously to the API software because of the extra delay and latency incurred when sending requests via logically and physically long paths, such as passing through the System HSA and cabling to and from the I/O cage.

1.4 Sharing the Crypto Express3 coprocessor between logical partitions

A Crypto Express3 coprocessor can be shared between 16 logical partitions, whatever its mode of operation (coprocessor or accelerator). Therefore, a single Crypto Express3 feature with two coprocessors (FC0864) can provide shared cryptographic services to up to 32 logical partitions.

Functional areas in the CEX3C coprocessor are physically dedicated to each logical partition that has been set up to share the coprocessor. These dedicated functional areas are the cryptographic *domains* and physically contain an isolated set of master keys (DES-MK, AES-MK, and ASYM-MK) for the use of the logical partition to which the domain is assigned.

There are 16 domains in the coprocessor (domains 0 to 15), and an active logical partition can use only one domain in a given coprocessor at any point in time, and the domain should not be already in use by another active logical partition. The coprocessor and the domain number in the coprocessor that the logical partition is to use are indicated in the partition's image profile.

See Chapter 4, "z10 Cryptographic configuration" on page 73, for further information about the logical partitions settings.

Note: The notion of logical partition sharing is not relevant for the CPACF, as the facility is available in its entirety to the logical partition currently being dispatched on the physical processing unit.

1.5 System z10 and Crypto Express3 configuration information

This section provides a summary of configuration information that pertains to the installation and use of the Crypto Express3 feature in System z10.

1.5.1 Maximum number of CEX3C features and coprocessors/accelerators

Table 1-2 summarizes the configuration data pertaining to the maximum number of CEX3Cs and CEX3As in a System z10.

Table 1-2 System z10 Crypto Express3 features configuration data

	Minimum number of features per System z10	Maximum number of features per System z10	Number of cryptographic coprocessors or accelerators per feature	Maximum number of cryptographic coprocessors or accelerators per System z10	Number of logical partitions that can access a Crypto Express3 coprocessor or accelerator
Crypto Express 3	0	8	2 or 1	16	16

Important: The system restricts sharing of coprocessors by logical partitions by applying the following rule: There can be only one active logical partition that can access a single [coprocessor number + domain number] combination.

The consequence of this is that giving access to CEX3C services to more than 16 active logical partitions entails using more than one single coprocessor.

1.5.2 Hardware feature codes referred to in this book

The cryptographic features and their feature codes that are available, as of the writing of this book, for System z10 G3 EC or BC are:

Feature code **Description**

3863 Crypto enablement FC. This feature code is installed during system manufacturing if ordered with the system, or it is delivered to the installation as a LIC DVD if ordered after system delivery.

The CPACF only provides the Secure Hash Algorithm (SHA) function if FC 3863 is not installed in the system, whereas it provides the full range of functions described below when the FC is installed. The Crypto Express3 coprocessor does not provide any service if FC 3863 is not installed.

0863 The Crypto Express2 feature that contains two coprocessors (CEX2C). The Crypto Express2 feature is now withdrawn from marketing and has been replaced by the Crypto Express3 feature.

0870 Crypto Express2-1P feature. The Crypto Express2 feature is now withdrawn from marketing and has been replaced by the Crypto Express3-1P feature.

0864 The Crypto Express3 feature that contains two coprocessors (CEX3C). Also re-configurable as a Crypto Express3 Accelerator (CEX3A). The Crypto Express3 feature is installable in the system z10 EC and BC.

0871 Crypto Express3-1P feature that contains one coprocessors (CEX3C). Also re-configurable as a Crypto Express3 Accelerator (CEX3A). The Crypto Express3-1P can only be installed in a System z10 BC.

0840 TKE V6 workstation hardware with Ethernet connection: Up to 10 TKE workstations can be ordered with a System z10. This includes the

workstation hardware itself with an Ethernet connection, DVD drive, and flat panel monitor.

0858 TKE 6.0 LIC DVD.

0885 TKE smart card reader.

0884 TKE additional smart cards.

Important: The TKE 6.0 workstation can be used to manage the cryptographic coprocessors on an IBM eServer™ z890, IBM eServer z990, IBM System z9®, or IBM System z10. It cannot be used to manage the cryptographic technologies used on IBM eServer 800 and 900 systems and their predecessor systems.

TKE versions previous to V6 cannot be upgraded to the TKE V6 hardware. However, data migration aids are provided with TKE Version 6.

1.5.3 The TKE workstation feature

A TKE workstation is an optional priced feature that allows managing, from a central point of control, master keys and operational keys in Crypto Express3 coprocessors that are in use by z/OS or by Linux for System z programs. The TKE creates a highly secure logical channel via TCP/IP communications between the workstation and the CEX3C coprocessors that it manages, that is used to exchange sensitive data, over TCP/IP, such as the value of the master key to be set or coprocessor status information. The security of the logical channel relies on the use of strong encryption and digital signature of the exchanged information.

Note: The TKE communicates directly with the target coprocessor. Encryption and decryption of sensitive information, along with their proper authentication via digital signature, occurs in the CEX3C coprocessor itself. What can be captured on the network or in system's storage is encrypted communication packets only.

For installations that did not elect to use a TKE workstation, a subset of the TKE management functions is still available via interactive facility: TSO/E ISPF panels in z/os and command-line interface (CLI) in Linux for System z. These functions are the ones deemed necessary for day-to-day operations but, in this case, do not benefit from the level of security that is achieved using a TKE workstation.

Figure 1-3 is an illustration of a system configuration where the coprocessor API is provided by the ICSF component of z/OS and the optional TKE workstation is used for key administration in the CEX3C coprocessors.

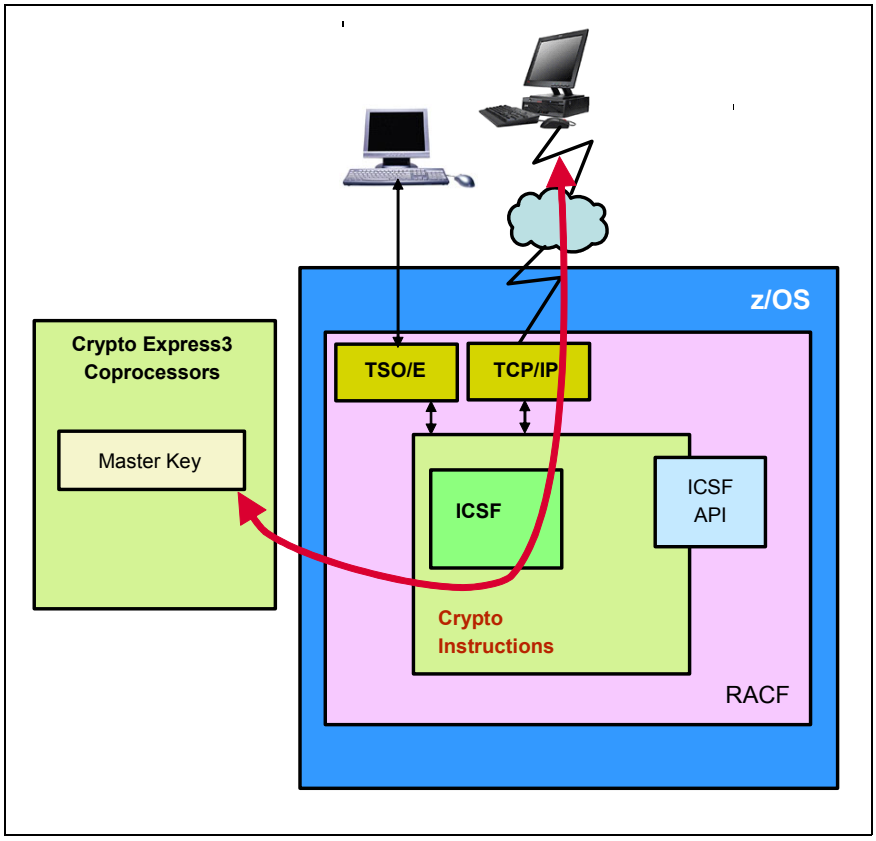


Figure 1-3 The trusted key entry (TKE) workstation and ICSF

The secure channel can be established from a single TKE workstation to several CEX3C coprocessors hosted in a single or different physical systems, provided that these systems are executing the TKE supporting software and have proper TCP/IP connectivity with the workstation.

A TKE workstation can be set up so that sets of coprocessors can be defined and handled as groups, as the same TKE initiated functions are automatically repeated on each member of the group.

Using a TKE workstation also brings increased security for coprocessor administration in that the security officers in charge are identified and authenticated by the CEX3C coprocessor with their personally assigned digital signature. The TKE also offers options that require multiple security officers to be present and to provide approval before performing selected coprocessor administrative functions.

Among the functions that only the TKE optional feature can provide (as opposed to the functions that in the z/OS ISPF panels or the Linux CLI) is the capability of enabling or disabling specific elementary functions in the cryptographic coprocessors hardware by acting on the coprocessor access control points (ACPs).

More details on the TKE optional feature are given in Chapter 7, “Trusted Key Entry” on page 173.

Important: TKE operations are not relevant to the CPACF or the Crypto Express3 coprocessor operating in accelerator mode (CEX2A), as these devices do not exploit master keys and secure keys.

1.6 Cryptographic features comparison

This section provides a comparison matrix for readers to easily assess the functional differences between the CPACF and the CEX3C coprocessor.

1.6.1 Quick summary of the main functional differences between CPACF and CEX3C coprocessor

The main differences are listed below:

- The CPACF:
  - Does not support secure keys (that is, keys encrypted under a master key that itself resides in the coprocessor). It does however support, beginning with System z10 G3, the use of protected keys where the key-encrypting-key is kept in non-addressable system storage.
  - Does not support asymmetric algorithms.
  - Is not sharable between logical partitions, as it is available in its entirety to the PU on which the logical partition has been dispatched.
  - Provides very high-speed synchronous hardware services in all cases.
- The CEX3C coprocessor:
  - Supports secure keys for both symmetric and asymmetric algorithms.
  - Supports an asymmetric (RSA) algorithm.
  - Can be shared by up to 16 logical partitions.
  - Can be manually switched into accelerator mode for the purpose of providing very high-speed hardware assist to the SSL/TLS protocol handshake.
  - Always runs asynchronously to the software providing the API.
  - The speed achieved for symmetric-algorithm-based functions does not match the CPACF performance. However, maximum security is achieved by the use of secure keys. If the use of secure keys is not an absolute requirement, installations might consider using the CPACF with protected keys for an optimum trade-off between performance and security.

Table 1-3 summarizes the functions and attributes of the cryptographic hardware features on System z10.

Table 1-3 System z10 cryptographic features comparison

Functions or attributes	**CPACF**	**CEX3C** (1)	**CEX3A** (1)
Supports z/OS applications using ICSF	**X**	**X**	**X**
Symmetric encryption and decryption using a secure key		**X**	
Symmetric encryption and decryption using a protected key	**X**(1)		

Functions or attributes	CPACF	CEX3C (1)	CEX3A (1)
Symmetric encryption and decryption using clear key only	X(1)		
Asymmetric encryption (RSA) using secure or clear key		X	
Provides highest SSL handshake performance (clear key only)			X
Provides highest symmetric (clear key) encryption performance	X(1)		
Provides highest asymmetric (clear key) encryption performance			X
Disruptive process to enable			
Requires IOCDS definition			
Uses CHPID numbers			
Is assigned PCHIDs		X (4)	X (4)
Functionally embedded in each PU	X		
Requires CPACF Enablement FC 3863	X(3)	X(6)	X(6)
Requires the use of the operating system API		X	X
Offers user-defined extension function support (UDX)		X	
Usable for data privacy - encryption and decryption processing	X(1)	X	
Usable for data integrity - hashing and message authentication	X	X	
Usable for financial processes and key management operations		X	
Crypto performance RMF™ monitoring		X	X
Requires master keys to be set		X(5)	
Retained keys support		X(2)	
Tamper-resistant hardware packaging		X	X
High-performance SHA-1and SHA-2 hash functions	X		
Pseudo random number generator	X		
Random number generator		X	
Clear key RSA		X	X
Double length DUKPT support		X	
Europay Mastercard VISA (EMV) support		X	

Notes

The following notes pertain to Table 1-3 on page 16.

1. Requires CPACF enablement (that is, FC 3863) to be installed.
2. IBM does not suggest using retained keys in System z.
3. Using the CPACF for DES, T-DES, or AES encryption requires the CPACF to be enabled, even when invoked from Linux.
4. CEX3C/CEX3A is assigned two PCHIDs per feature (one per coprocessor or accelerator).

5. The master key is not required to be set for the CEX3C to produce random numbers or to perform clear key RSA operations.
6. The CPACF enablement is not required for Linux if only the RSA clear key operations, provided by CEX3C/CEX3A, are being used.

1.6.2 Performance data

IBM regularly publishes cryptographic performance data for z/OS or Linux for System z based on measurements done by IBM using test programs calling the CPACF or CEX3C services. The obtained figures are published at:

http://www.ibm.com/servers/eserver/zseries/security/cryptography.html

1.7 Application programming interfaces

APIs are provided in the IBM mainframe operating systems for programs to invoke the hardware cryptographic functions. In this section we briefly describe these APIs. Note that the z/OS Integrated Cryptographic Service Facility and the Linux for System z APIs are discussed in further detail in this book in Chapter 5, “Integrated Cryptographic Service Facility (ICSF): Recent changes” on page 83, and Chapter 8, “A brief overview of integrated hardware cryptography implementation in Linux for System z” on page 229.

1.7.1 The z/OS API - ICSF

The Integrated Cryptographic Service Facility (ICSF) is a base component of z/OS that implements the IBM proprietary Common Cryptographic Architecture (CCA) API.

The CCA API is the lowest available API available in z/OS for programs to invoke Crypto Express3 services. The CPACF functions can also be invoked through the CCA API by programs that do not implement the MSA instructions.

ICSF overview

This section describes the overall hardware and software layout of the hardware cryptography implementation in System z10 with z/OS. Figure 1-4 illustrates the overall z/OS hardware cryptography support infrastructure and the pivotal role of ICSF. Details about ICSF operations can be found in Chapter 5, “Integrated Cryptographic Service Facility (ICSF): Recent changes” on page 83.

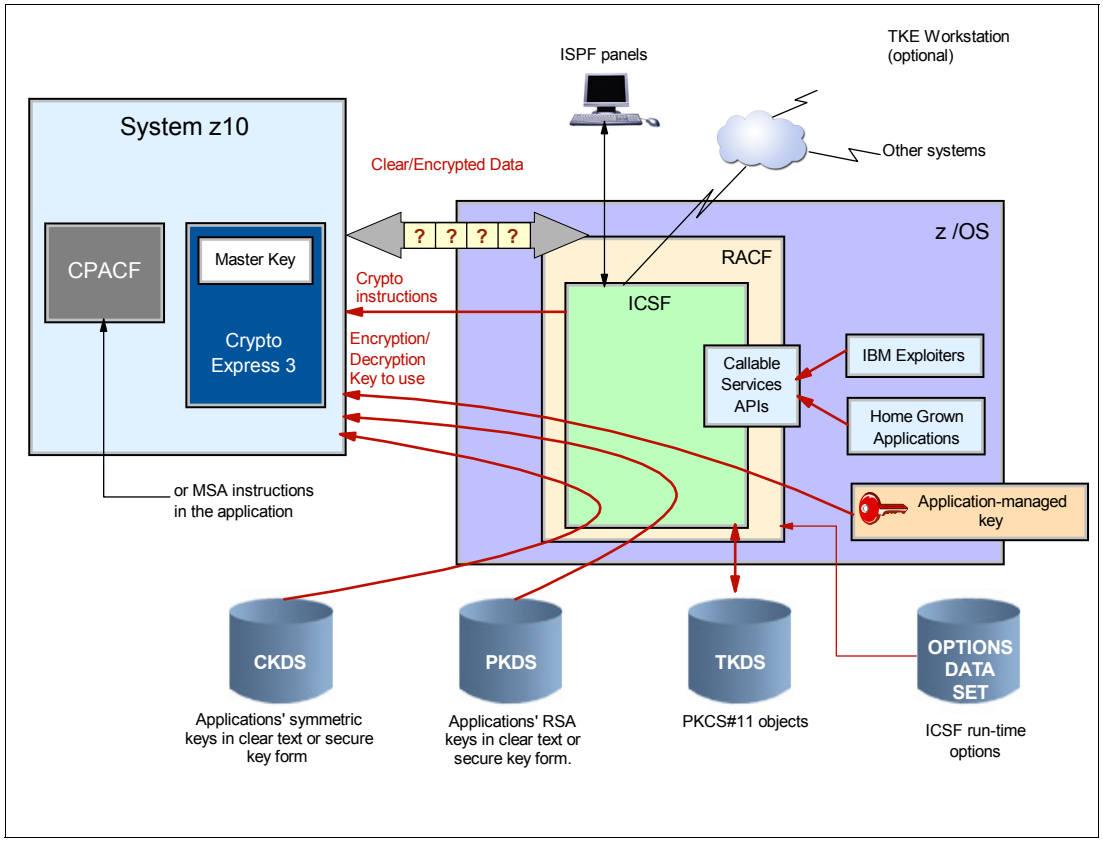


Figure 1-4 Overall hardware and software layout with z/OS

Important: It is important to understand that ICSF, as it is executing in the z/OS logical partition, will only “see” the CEX3C/CEX3A domain assigned to this partition. Note that ICSF can “see” one or more CEX3C/CEX3A coprocessors depending on how available coprocessors are allocated to the system’s logical partitions. However, it will always “see” the same single domain in each one of these coprocessors. For further information refer to the explanation on logical partition setup in Chapter 4, “z10 Cryptographic configuration” on page 73.

A typical System z cryptographic usage example, using Figure 1-4 as a reference, is as follows:

- The exploiters of the cryptographic services call the ICSF CCA API. Some functions are performed by the ICSF software without invoking any cryptographic coprocessor. Other functions result in ICSF executing routines containing the cryptographic instructions that drive the CPACF or the CEX3C/CEX3A, depending on what the requested function is. Whenever more than one CEX3C/CEX3A is available to ICSF, ICSF proceeds with workload dispatching by driving the incoming request to an idle coprocessor or by queueing the request to the best potential candidate if no coprocessor is currently idle. The cryptographic instructions that are used to interface with the CPACF are publicly

disclosed in *z/Architecture Principles of Operation*, SA22-7832. The cryptographic instruction set that is used to drive functions execution in the CEX3C/CEX3A is not disclosed by IBM.

- The cryptographic coprocessor is provided with the following:
  - The data to encrypt or decrypt, from the application's storage.
  - The key used to encrypt or decrypt. ICSF obtains the key from the application or the key can be stored in an ICSF-managed VSAM data set and pointed to by the calling application (which then provides the label that the key is stored under). The Cryptographic Key Data Set (CKDS) is used to store the symmetric keys, and the Public Key Data Set (PKDS) is used to store the asymmetric keys. The stored keys can be secure keys or clear text keys, as dictated by the specifications of ICSF callable services. Applications have also the capability of managing their own key repositories, on files or in storage, also in secure or clear text form.
  - The Token Key Data Set (TKDS) is an optional data set that is used by ICSF only for storing z/OS PKCS#11 persistent tokens and objects.

Note that for high-speed access to the keys in the VSAM data sets, the data set contents are duplicated in ICSF-owned data space at ICSF startup.

PKCS#11 API and Elliptic Curve Cryptography support

The PKCS#11 cryptographic API support in z/OS has been introduced in z/OS V1R9 with ICSF HCR7740 to provide an alternative to the IBM Common Cryptographic Architecture API to broaden the scope of cryptographic applications that can make use of System z integrated hardware cryptography. This API was initially provided in the C language support in z/OS, which, transparently to the C application programs, was calling ICSF services for CCA functions that can contribute to PKCS#11 computations.

With ICSF HCR7770, new ICSF callable services have been added so that non-LE programs can invoke the PKCS#11 cryptographic functions. These services are described in Chapter 6, "z/OS support for the PKCS#11 API" on page 121.

This is also the first time that Elliptic Curve Cryptography support is provided in z/OS.

Elliptic Curve Cryptography (ECC) is one of the many encryption algorithms that the z/OS PKCS#11 API supports. Elliptic Curve Cryptography is an emerging public-key algorithm to eventually replace RSA cryptography in many applications. ECC is capable of providing digital signature functions an key agreement functions. The new ICSF services provide ECC key generation and key management and provide digital signature generation and verification functions compliant with the ECDSA method described in ANSI X9.62 ("Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)." ECC uses keys that are shorter than RSA keys for equivalent strength. RSA becomes impractical at key lengths that would be equivalent to AES-192 and AES-256 key strength.

Be aware that, for the initial implementation of the PKCS#11 API, these services exploit, whenever possible, the existing underlying CCA services, which provide hardware assist from the CPACF and the CEX3C. However, ICSF also hosts a pure software implementation of these services so that users can still call them even if there is no cryptographic hardware available in the system.

Access control to cryptographic resources and function

Note also that SAF/RACF plays a fundamental role in securing access to the z/OS resources that pertain to the use of hardware cryptography.

As described later in this book, z/OS ICSF manages the security of these resources referring to RACF profiles defined in the following classes:

- CSFSERV
- CSFKEYS
- XCSFKEY
- CRIPTOZ
- FACILITY
- XFACILIT

ICSF releases

As of the writing of this book, the ICSF release delivered with z/OS V1R11 is FMID HCR7751. The new ICSF release (FMID HCR7770) has been made available as an SMP/E installable web deliverable (“Cryptographic Support for z/OS V1R9-V1R11”) at the following website:

`http://www.ibm.com/eserver/zseries/zos/downloads/`

This book refers to ICSF operations as implemented in ICSF HCR7770.

For a list of ICSF versions and FMID cross-references, see the IBM Techdocs webpage at:

`http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/TD103782`

1.7.2 Hardware cryptography exploitation in z/VSE

The main use of hardware cryptography in z/VSE stemmed originally from the need to provide hardware assistance to the SSL/TLS protocol. To meet this objective, specific cryptographic services were developed to be used internally in z/VSE. A decision was later made to externalize these services as the set of APIs shown in Figure 1-5 on page 22. Note that, because of the initial intent of providing cryptographic support specifically for SSL/TLS, the z/VSE cryptographic APIs are hosted by the z/VSE TCP/IP stack component. The z/VSE TCP/IP stack provides the cryptographic services and transparently invokes hardware cryptography when relevant and available. Hardware cryptography using the CEX3C/CEX3A and CPACF devices is supported starting with TCP/IP for VSE 1.5D running on z/VSE 4.2.

RSA acceleration

Note that z/VSE exploits the CEX3C or CEX3A for RSA acceleration only, that is, using clear keys for:

- RSA decryption or encryption operations during initiation of the SSL/TLS session
- RSA signature of certificates built by the z/VSE certificate utility

CPACF exploitation

The CPACF is used for the SSL/TLS data transfer when one of the following symmetric algorithms has been agreed upon between the client and server:

- DES
- Triple DES
- AES-128
- 192
- 256

The CPACF is also used when the SHA-1 or SHA-2 algorithm has been selected for data integrity checking, both during data transfer or when building a certificate via the certificate utility.

Note: Whether or not hardware cryptography is used is transparent to the z/VSE TCP/IP applications. Whenever hardware cryptography cannot be used, then the software implementation of the cryptographic services in the TCP/IP stack is used.

The use of hardware cryptography can be disabled via a setting in the $SOCKOPT Phase for the z/VSE TCP/IP stack.

Infrastructure

The lowest-level hardware cryptography API provided in z/VSE is the CryptoVSE API, where applications or middlewares can call specific cryptographic services that are eventually performed by the CPACF or CEX2C (in coprocessor or accelerator mode) devices.

The SSL for VSE API is a higher-level API, similar to the z/OS System SSL API, intended to provide SSL/TLS-enabled applications with runtime assistance to handle the SSL/TLS protocol. This API transparently invokes the CryptoVSE functions.

The SSL Daemon (SSLD) is not an API per se, but manages the SSL/TLS protocol on behalf of non-SSL-enabled applications, much alike z/OS Application- Transparent TLS (AT-TLS) does.

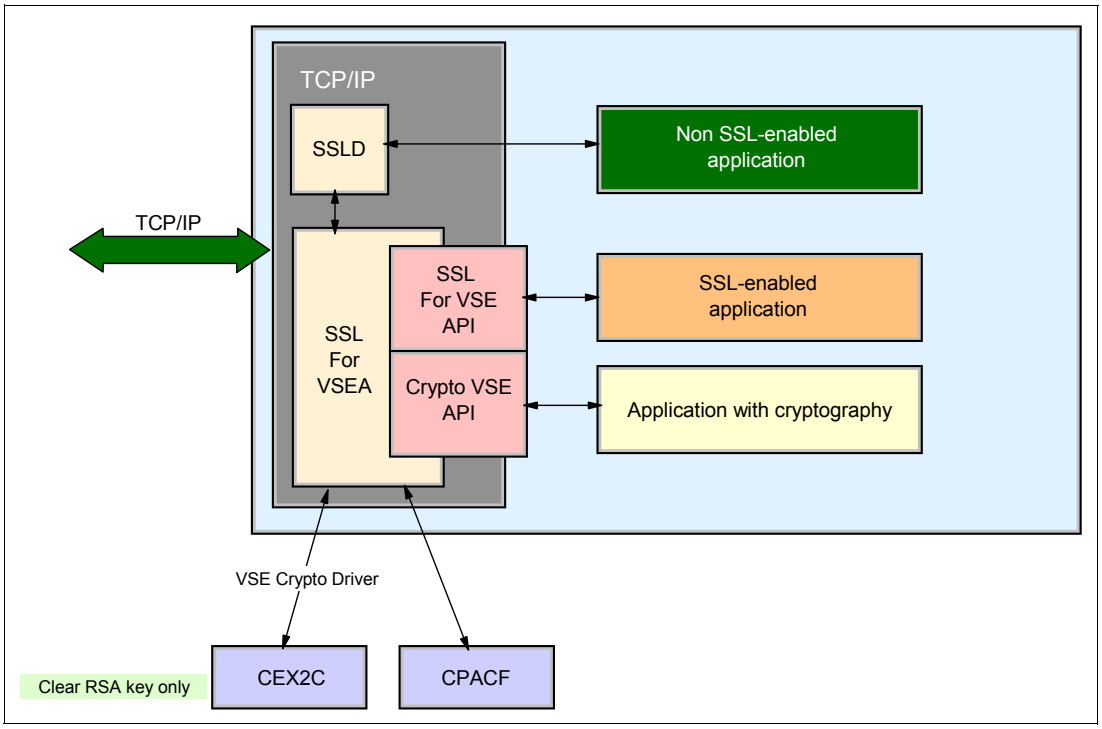


Figure 1-5 z/VSE hardware cryptography infrastructure

1.7.3 Hardware cryptography exploitation in Linux for System z

The hardware cryptography infrastructure for Linux for System z is shown in a very simplified form in Figure 1-6.

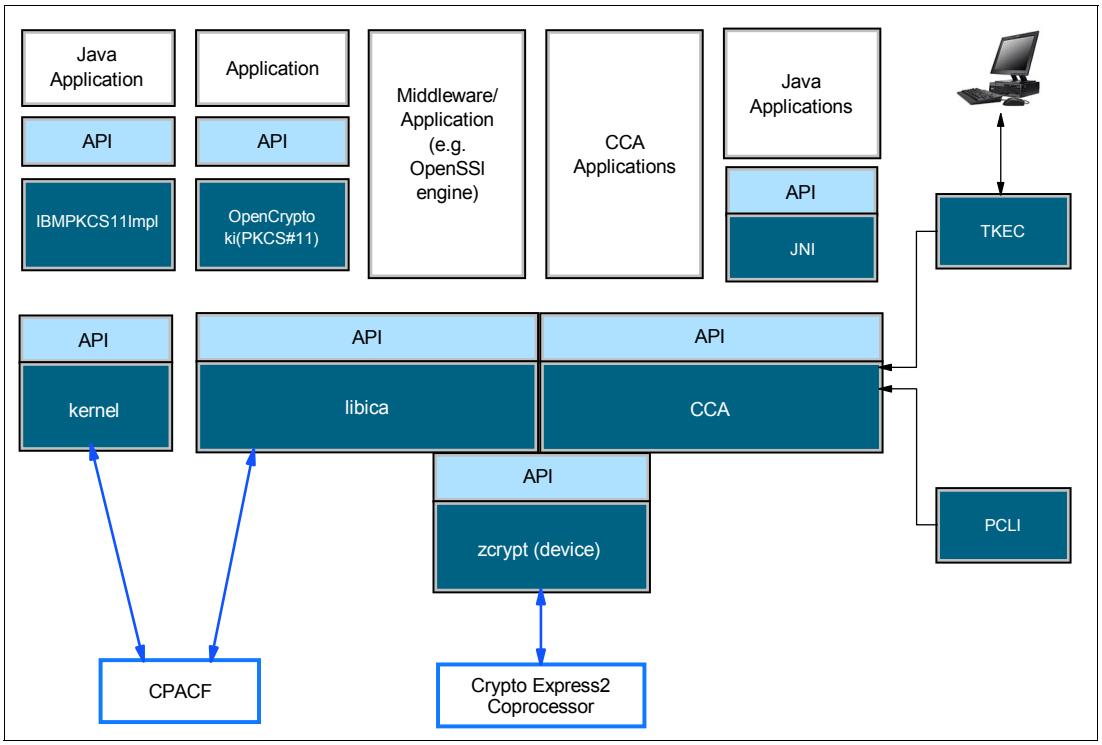


Figure 1-6 Linux for System z hardware cryptography infrastructure

There are globally three layers of APIs provided for Linux for System z hardware cryptography exploitation:

- At the lowest level, the *zcrypt* device driver provides an API usually not exploited directly by applications but rather intended for intermediate software layers that in turn provide more sophisticated cryptographic functions to the next upper level of code. zcrypt drives the CEX3C coprocessor.
- The intermediate level consists of:
  - The libica cryptographic functions library, which offers a wide range of cryptographic functions, some of them being transparently performed by the hardware cryptographic devices under control of the device driver. The libica functions operate with clear keys only.
  - Certain cryptographic APIs are also available in the Linux Kernel, which also operate with clear keys only.
  - The IBM CCA API is also provided for Linux for System z. Among other things, this API provides Linux applications with the capability of using secure keys and now CPACF protected keys as well. This API comes along with coprocessor management facilities provided via the command-line interface or via the TKE workstation.
- At the top, popular high-level APIs such as the PKCS#11 API, or its JAVA version, the IBMPKCS11Impl cryptographic API, are exploiting the functions provided by the lower levels.

  Note also the Java™ version of the CCA API that exploits the JNI to invoke CCA services.

More details on the hardware cryptography infrastructure in place in Linux for System z are given in Chapter 8, “A brief overview of integrated hardware cryptography implementation in Linux for System z” on page 229.

1.7.4 Setting up the hardware cryptography configuration of z/VM

In z/VM, real cryptographic coprocessors are assigned to guest virtual machines by specifying the CRYPTO parameter in the VM guest machine directory with the DOMAIN and APDEDICATED or APVIRTUAL operands. Refer to *z/VM CP Planning and Administration*, SC24-6083, for further explanations on the use of these parameters:

- APDEDICATED

  This specifies the numbers of the APs (that is, the CEX3C coprocessors) that the virtual machine can use for dedicated access. The DOMAIN operand also must be specified to indicate the coprocessor domain to access in the APs. The APs specified must be selected from the set of APs selected on the PCI Cryptographic Online List on the Crypto Image Profile Page for the z/VM logical partition, or from the candidate list provided that the APs have been brought online to the z/VM logical partition. The DOMAINs specified must be part of the set of domains selected in the Usage Domain Index list in the Crypto Image Profile Page for the Logical Partition.

- APVIRTUAL

  This tells the z/VM Control Program that this virtual machine might share access with other virtual machines to the clear-key functions only of the CEX3C coprocessor. In that case, z/VM drives the requests to whatever coprocessor is available, and it is not necessary to specify the DOMAIN operand when specifying the APVIRT operand, as z/VM rejects all requests for services that would involve the use of secure keys.

Figure 1-7 is an example of a VM guest machine setup, where the z/VM system logical partition has access to APs 5 and 6 in coprocessor mode and AP 2 in accelerator mode. The z/OS VM guest machine's directory has been set up to specify a dedicated access to domain 1 of AP 5 and 6, and the Linux VM guest machines are sharing access to whatever accelerator is available to the z/VM system.

There is no setup required to share the CPACF, as the facility is available to whichever guest program is dispatched on the PU.

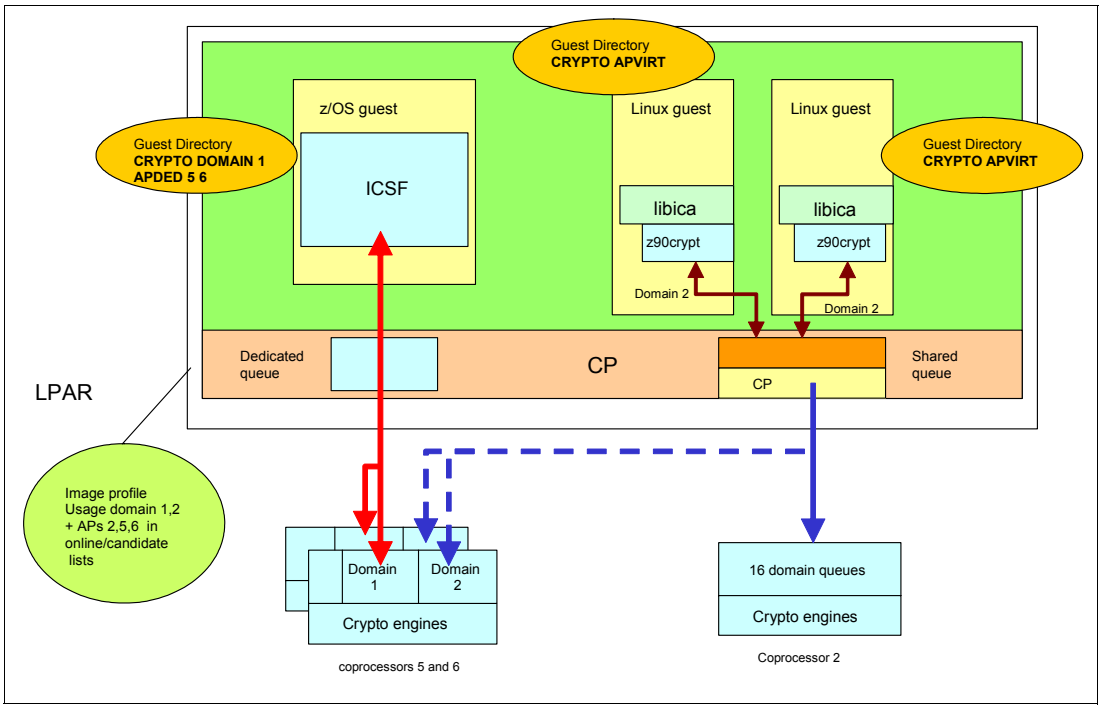


Figure 1-7 Hardware cryptographic coprocessors assignment to VM guest machines

1.8 Required software levels

We indicate in this section what software levels are required in order to have the full range of hardware cryptographic services to be available in a System z10 installation.

1.8.1 Support for System z10 CP Assist for Cryptographic Function

Table 1-4 provides the operating systems levels where the CPACF is recognized and can be exploited through APIs provided in the operating system (the latter does not apply to z/VM, as z/VM does not provide an API).

Table 1-4 Support requirements for enhanced CPACF

Operating system	Support requirements
z/OS[a]	z/OS V1R7 and later: The function varies by release. Protected key requires z/OS V1R9 and later, plus PTFs.
z/VM	z/VM V5R3 and later: Supported for guest use. Protected key not supported.
z/VSE	z/VSE V4R1 and later, and IBM TCP/IP for VSE/ESA V1R5 with PTFs.

Linux on System z	Novell SUSE SLES 9 SP3, SLES 10 and SLES 11 Red Hat RHEL 4.3 and RHEL 5 The z10 EC CPACF enhancements can be used with: ► Novell SUSE SLES 10 SP2 and SLES 11 ► Red Hat RHEL 5.2
TPF and z/TPF	TPF V4R1 and z/TPF V1R1

a. CPACF is also exploited by several IBM Software product offerings for z/OS, such as IBM WebSphere® Application Server for z/OS.

1.8.2 Crypto Express3

The Crypto Express3 support is not provided by the same component in all IBM mainframe operating systems. For instance, the ICSF component of z/OS provides this support, whereas it is provided by TCP/IP in z/VSE. Table 1-5 lists the minimum software level required to support the Crypto Express3 feature when configured either as a coprocessor or an accelerator.

Table 1-5 Crypto Express3 support on z10 EC

Operating system	Crypto Express3 support
z/OS and ICSF	V1R11: Web deliverable(1) V1R10: Web deliverable (1) V1R9: Web deliverable (1) V1R8: Not supported V1R7: Not supported
z/VM	V5R3: Service required, supported for guest use only
z/VSE	V4R2 with IBM TCP/IP for VSE/ESA V1R5, service required
Linux on System z	Note (2) Novell SUSE SLES 11 Novell SUSE SLES 10 SP3 Red Hat RHEL 5.4
TPF V4R1	Not supported
z/TPF V1R1	Service required (accelerator mode only)

[1]ICSF web downloads are available at http://www-1.ibm.com/servers/eserver/zseries/zos/downloads.

[2] Support for Crypto Express3 is provided at the same functional level as for Crypto Express2.

2

More on the 4765 Cryptographic Coprocessor and the Crypto Express3 feature

Information pertaining to the 4765 Cryptographic Coprocessor and the Crypto Express3 feature core elements of System z hardware cryptography implementation has been given in 1.3, “Overview of System z10 hardware cryptographic features” on page 4.

This chapter provides an additional information about the operational characteristics of the IBM 4765 Cryptographic Coprocessor and the Crypto Express3 feature.

2.1 IBM hardware cryptographic coprocessors design points

The experience gained during the past two decades with hardware cryptographic products, either to be integrated in mainframes or installed in smaller and distributed systems, led IBM to ensure that the issues discussed in this section were addressed when designing new hardware cryptographic coprocessors.

2.1.1 Flexibility

Because of the rapid evolution in user demand for cryptographic functions and the techniques that become available to fulfill this demand, we have seen that flexibility is an essential feature to be addressed in the implementation of cryptographic coprocessor functions. Flexibility in this context entails providing the following capabilities:

- It must be easy to add new functions to the standard set already existing in the coprocessor base when the coprocessor is already in operation at a customer site.
- It must be possible to answer specific user requirements with customized functions that then coexist with this standard set.

This translates into a more specific requirement of being able to update the coprocessor firmware at the customer's site, in a very secure manner that does not degrade the operational integrity and security level of the device.

2.1.2 Performance

There is always been a requirement for high-performance symmetric-key cryptography. As the requirements for asymmetric cryptography-based functions has dramatically increased in recent years, there is also a very noticeable increase in performance requirements for these algorithms as well, as they rely on complex mathematical operations and use very large numbers.

2.1.3 Acceleration and security

Acceleration and security are two fundamentally different goals when implementing cryptographic functions:

- Acceleration, as the name implies, consists in simply accelerating the complex cryptographic algorithms execution to increase system throughput beyond what is possible when the algorithm are executed in by software only. In this environment, the design priority is given to the implementation of whatever can contribute to increasing the coprocessor throughput.
- The design approach might consider a high security level in the coprocessor operations to be the primary goal. Although high performance also remains a very important factor, it is essential in this environment that all cryptographic keys be securely protected from disclosure, modification, and misuse. It is equally important to guarantee that there can be no tampering with the security functions embedded in the coprocessor operations.

  These requirements dictate design priorities and trade-offs other than those with acceleration only so that potential interferences with pure performance are kept to a minimum.

2.1.4 Multiplatform support and interoperability

Today, many classes of cryptographic applications are intended to be run on different hardware platforms, still with the requirement of providing hardware implemented acceleration and security. In most of these cases users also place a high value on keeping the cryptographic functions compatible and interoperable across all of their platforms.

This adds the requirement of supporting industry-standard cryptographic APIs (for example, PKCS#11) and, whenever necessary, making more specific APIS available on several platforms (as the IBM CCA API is today).

2.1.5 Export control

Although in recent years export controls on cryptographic products have been almost entirely removed, and the need for special export control features has been greatly reduced, government rules can change at any time. It is still important to design all products so that they can accommodate new regulations that might be introduced.

2.1.6 Security certifications

Certification of cryptographic devices to the appropriate security standard by an independent expert organization has become mandatory over the years. This is mainly due to the depth of knowledge that is required today to properly assess the security of cryptographic products.

One of the most important contemporary standards related to cryptographic coprocessors is NIST FIPS 140, a standard for evaluating the physical and logical security of a hardware-based or software-based cryptographic module. The current level of the standard is FIPS 140-2.

This is a well-known standard and has been scrutinized by experts in the field for years. Certification under that standard provides a high degree of assurance that the product will offer good security in the area covered by the standard.

Note: The predecessor device to the IBM 4765 coprocessor, the 4764-001 Cryptographic Coprocessor, has been certified at FIPS 140-2 Level 4 (the highest certification for the FIPS 140-2 standard).

As of the writing of this book, the IBM 4765 is under evaluation for FIPS 140-2 Level 4 certification.

2.2 The 4765 Cryptographic Coprocessor

The 4765 is the follow-on product to the IBM 4764-001 Cryptographic Coprocessor (also known as PCI Extended Cryptographic Coprocessor (PCIXCC), which was the core element of the Crypto Express2 feature. This section describes the functional characteristics of the 4765 coprocessor and explains how it differs from its 4764-001 predecessor.

2.2.1 Coprocessor principles of operation

Figure 2-1 illustrates the coprocessor implementation within a host system. The host system communicates with the coprocessor over a PCI Express (PCIe) electrical interface using an architected data transport structure called a Connectivity Programming Request/Response Block (CPRB).

The CPRB is prepared by the IBM Common Cryptographic Architecture (CCA) API in the host system in response to a cryptographic application that requests execution of a CCA cryptographic function, also called a CCA *verb*.

The CPRB is sent to the coprocessor along with the verb parameters and data to process.

The operations in the coprocessor are managed by an operating system, actually a modified version of Linux, and performed by the CCA application software in the coprocessor. The CCA application software interprets the requests in the CPRB and issues new requests at a lower functional level to be executed by the coprocessor hardware and firmware facilities. Note that by design concurrent executions can occur in the coprocessor with multithreading support at the operating system and CCA application level.

Note also that in most cases the CCA verb is completely implemented in the coprocessor itself (that is, the API is just in charge of collecting and formatting the parameters to be sent, along with the verb identity to the coprocessor). There are very few occurrences of CCA verbs where the CCA API has to break down the application's request into more elementary requests to be executed by the coprocessor.

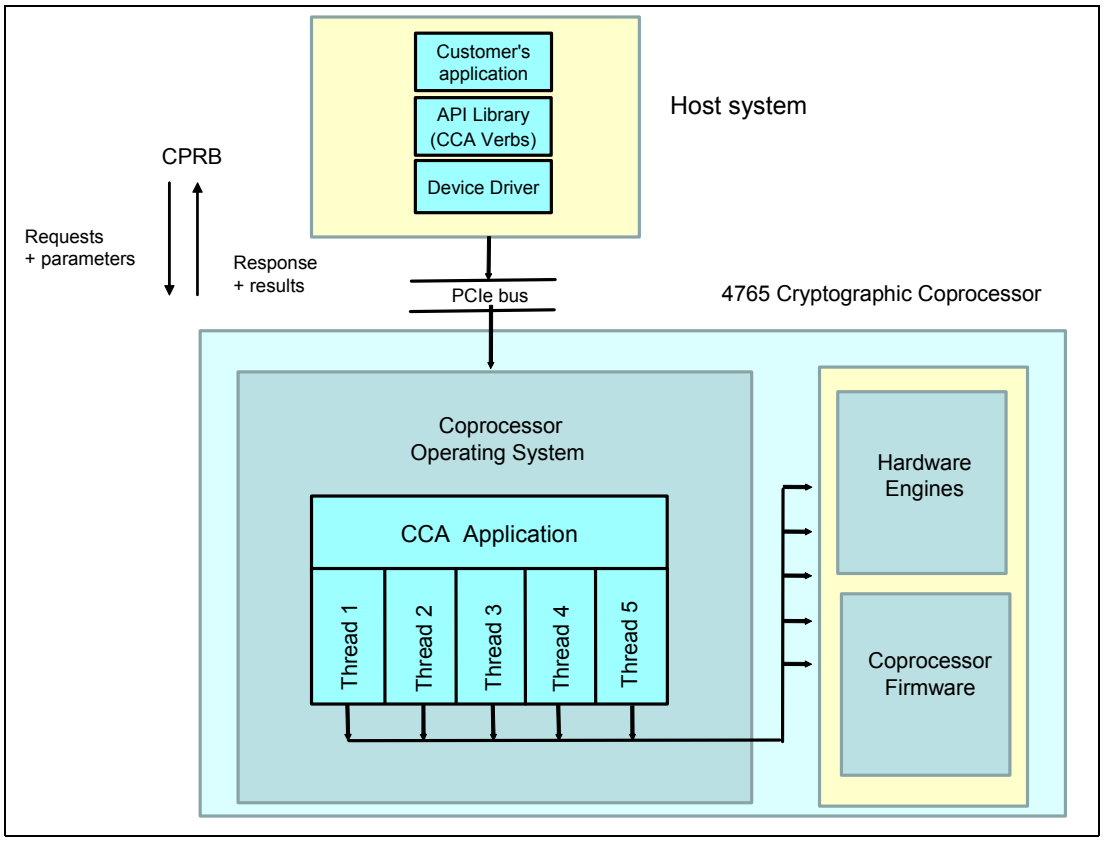


Figure 2-1 4765 Cryptographic Coprocessor operations flow

2.2.2 Hardware overview

The 4765 hardware is implemented in the form of a PCIe adapter card, with a secure module containing all security-related components. The module is designed to meet the stringent security requirements of the FIPS 140-2 standard at Level 4. The coprocessor hardware functional blocks are shown in Figure 2-2.

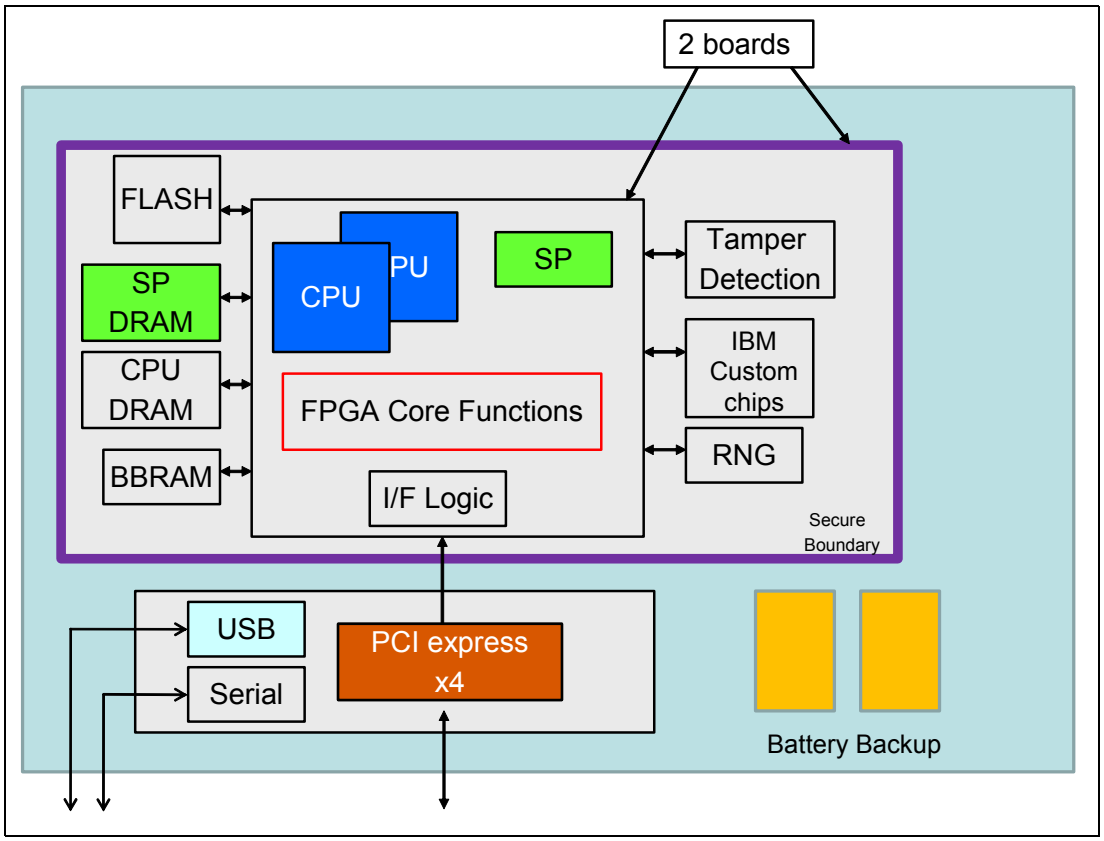


Figure 2-2 4765 Cryptographic Coprocessor functional blocks

The hardware components include:

- A PCI-e interface with four lanes (PCI-e X4).
- A USB and serial interface, which are not used when the card is plugged into a System z host.
- A dual CPU subsystem consisting of two IBM PowerPC® 405D5 RISC microprocessors operating at 400 MHz. They serve as the primary controllers of card operation with the execution of the operating system and the CCA application software. They orchestrate the operations of the special-purpose hardware in the card and implement communications with the host CCA API.

  The dual microprocessor scheme has been implemented to achieve maximum error detection in RISC processes by having tasks dispatched on both microprocessors, with the results compared in a lock step mode.

  The coprocessors share a 64 MB DRAM.
- Sixteen MB of flash-erasable programmable read-only memory (flash EPROM) for persistent data storage, and 4 MB of static complementary metal oxide semiconductor (CMOS) RAM backed up with battery power when the card is powered off (battery backed

up RAM, BBRAM). The secret values that the coprocessor protects are kept stored in the BBRAM.

- A dual service processor (SP) implementation for self testing of the card and code loading control. The dual SP implementation allows for concurrent patch and code update.

  Note: the 4765 concurrent patch and code update capability requires relevant support to be implemented in the host system's maintenance procedures and supporting software.

- An IBM-developed custom cryptographic chip that provides fast hardware implementation of the essential cryptographic algorithms used by the 4765 card. The chip is divided into two cryptographic algorithm sections:
  - The symmetric-key cryptography and hashing unit that provides the DES, TDES, and AES symmetric encryption algorithms and the SHA-1, SHA-256, and MD5 secure hashing algorithms. In addition to data encryption, the DES implementation includes both single-DES and TDES MAC support, conforming to ANSI X9.9 and ANSI X9.19.
  - The public-key unit provides the modular math functions that constitute the core functions used in asymmetric algorithms such as RSA, Diffie Hellman, or ECC.

  The chip has an interface to the hardware random-number source.
- A hardware-based cryptographic-quality random number source. The entropy is obtained from electrical noise from a semiconductor junction.
- A field-programmable gate array (FPGA) contains the logic for all interfaces between the host system, the PowerPC microprocessors, and the cryptographic chip. Because both the host system and the PowerPC microprocessors interface directly with the FPGA to talk to each other or to request cryptographic services, the FPGA is the key component for all internal and external programming interfaces.

  The FPGA provides two fundamentally different communication paths for host-to-card transactions:
  - Normal path
  - Fast path

  The fast path is a mode of operation that provides very high performance for public-key cryptographic functions. It gives the host system a direct hardware path to the chip's public-key cryptographic engine so that data does not have to stop in PowerPC memory and no software is involved. The fast-path design supports operations using cleartext RSA keys.

  Note: As of the writing of this book, the 4765 fast path design supports RSA keys up to 2048 bits long. 4096-bit key operations, even with clear text keys, have to be processed in the slower normal mode of operations.

Secure boundary and tamper detection

The secure packaging technology is designed to detect or prevent all known physical attacks that might allow determined adversaries to extract secret data or tamper with the execution of critical operations within the card.

The secure module on the 4765 card is designed with industry-leading tamper-detection features. The security related electronic components are wrapped in a flexible mesh with narrow, embedded, overlapping conductive lines that prevent any physical intrusion by drilling, mechanical abrasion, chemical etching, or other means.

If the conductive lines are damaged, it is sensed by circuits inside the module, and all sensitive data is immediately destroyed, or *zeroized*. This is done by zeroizing the battery-backed static RAM. All sensitive data is stored either directly in the static RAM or in flash memory and encrypted under a 168-bit TDES key that is itself stored in that static RAM. If that key is destroyed, all encrypted data in the flash memory is rendered unusable.

Other special circuits sense attacks that can cause imprinting in the static RAM. Imprinting is a process that can permanently burn data into the RAM, so that the same data appears each time the RAM chip is powered on. Different data can be written to the chip while it is operating, but the next time that it is powered on, the originally imprinted data appears again as the initial memory content.

Imprinting can be caused by exposing the memory to either very low temperatures or X-rays, and the tamper circuitry detects either of these and zeroizes the memory before imprinting can occur. Finally, there are attacks that are driven by manipulating the power-supply voltages to the card, and these conditions are also detected to prevent the attacks from succeeding.

The security architecture of the hardware complements the secure code loading design, and the combination of the two provides the features that support FIPS 140-2 Level 4 certification.

Note: tamper detection is always in effect in the 4765 card, even if no sensitive data are stored yet in the card's memory.

There are some gradations in the circumstances leading to zeroization of the coprocessor's secrets, along with the consequences of this zeroization:

- Sensitive data can be zeroized on request of the host system. They can then be later re-initialized under control of the host system.
- Sensitive data are zeroized when the card is re-plugged in after being removed from the host system (a condition detected by the *intrusion latch*). The sensitive data can then be re-initialized under control of the host system.
- Some tampering conditions are detected but proved, after zeroization, to be transient conditions (for example, voltages out of range). Sensitive data is zeroized but can be re-initialized under control of the host system.
- Real tampering conditions are detected. Sensitive data are zeroized and the 4765 card cannot be operated anymore.

2.2.3 Coprocessor software layout

The software that runs on the PowerPC 405D5 in the coprocessor is divided into four separate components, or *segments* (Figure 2-3).

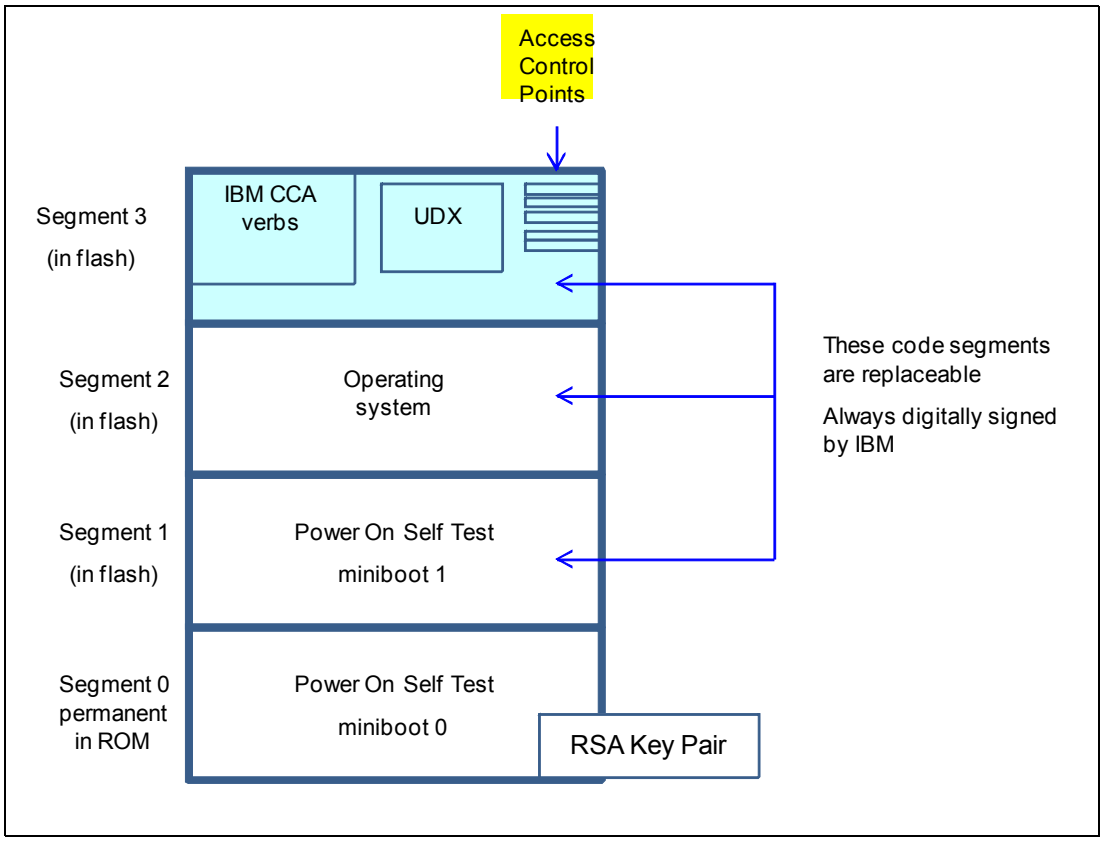


Figure 2-3 4765 software layout

Segment 0

The segment contains power-on self-test (POST) 0 and miniboot 0, stored in a region of flash EPROM that is unalterable once the card leaves the factory. POST 0 contains the small, low-level hardware self-test and setup.

Miniboot 0 is the lowest-level software for controlling the loading of software into segments 1, 2, and 3.

Note also that each 4765 is given a unique RSA key pair during its manufacturing process so that it can sign the data that it sends (this is required, for instance, in System z when communicating with the TKE workstation). The private key value is not known and the public key is bound to the specific 4765's unique serial number.

Segment 1

The segment contains POST 1 and miniboot 1. These are extensions to the POST and miniboot in segment 0, but have the important distinction that they can be securely reloaded after the card has been manufactured. Thus, segment 0 holds the minimum required POST and miniboot functions, while segment 1 contains the majority. This is done to minimize the chances that a critical error will occur in code that cannot be updated in the field.

Segment 2

The segment contains the operating system and device drivers. The 4765 card uses an open-source embedded Linux operating system. Special device drivers have been written to allow the operating system and application program to use the unique hardware inside the card.

Segment 3

The segment contains the application program that runs on the PowerPC 405 to give the card the cryptographic API functions seen by host programs. This application program currently implements the functions of the IBM CCA API. However, the card design allows you to expand the set of functions via User Defined Extension (UDX).

Note: Any operation of code reload or code update is performed under control of the card service processor.

2.2.4 Access control for 4765 card functions

The 4765 card has a built-in role-based access control mechanism. Facilities are provided for user identification and authentication and assignments of users to specific roles. Roles are granted access to the card functions. The access granularity to the card cryptographic functions is provided by the access control points, where each function has its own unique access control point to which a given role is permitted or not.

It is up to the host system implementation to decide whether the card role-based access control is to be fully exploited or to proceed instead with a subset of roles and access control points.

2.2.5 The CCA cryptographic application program

The IBM CCA functions are implemented in the 4765 card in the software that runs in segment 3. This is an application program executing under the embedded Linux operating system. The CCA software calls special Linux device drivers in order to access the cryptographic hardware and the host communications interface.

The CCA application is structured as a multithreaded application in which requests from the host are dispatched to one of several worker threads. This architecture allows the I/O (crypto and communication) to be overlapped by computations and provide a traditional blocking programming model. For example, some threads might be suspended waiting for RSA long-duration operations to complete, while others might be communicating with the host.

All the following principles of the IBM CCA design and operations are implemented in the 4765 CCA application program:

- The IBM CCA is a hardware-based cryptographic architecture. It is hardware-based in the sense that it must be implemented in a secure environment, where there is protection against disclosure or modification of secret CCA data and protection against modification of the execution of the CCA functions themselves.
- CCA must operate as a black box in which the inputs, outputs, and operations are well-defined in an external API, but where the data and functions within the box are protected against outside entities. The secure module on the card provides precisely the kind of protection needed to secure an API such as CCA.

  Many complex operations are thus implemented in CCA as atomic functions to ensure that no unprotected intermediate results ever appear outside the secure module.

- One of the security principles of CCA is that no application keys can ever appear in cleartext form outside the secure module. It is possible for a customer to load initial keys in cleartext form, but once those keys have been imported into the module, it is impossible to reveal them in cleartext form any more. Once keys are in the CCA environment, the API is designed to guarantee that they will always be securely encrypted.
- The preferred way to generate keys is to let the CCA module generate them so that they leave the module already in encrypted form, but this is not always practical when keys must be interchanged with non-CCA systems. To accommodate both this and legacy key-entry paradigms, CCA also provides functions to load DES keys in multiple cleartext key parts. These functions also provide dual control to ensure that different people are responsible for each of the separate key parts.
- In general, under CCA, all application keys are encrypted under a TDES or AES master key held securely inside the protected module. The keys can then be stored outside the module without having to be concerned about their security. They cannot be attacked because the master key used to encrypt them is itself secure inside the tamper protected module and will be zeroized if there is any attempted attack. This allows the customer to store a number of application keys, with the number limited only by storage on hard disk or other media.
- There is one exception to the application keys stored outside the secure module: CCA supports retained RSA keys, in which the RSA key pair is generated inside the secure module, and only the public key is ever allowed to leave the secure environment. The private key remains inside the module and is never allowed to leave in any form. This is designed to meet the strict demands of some standards, which require assurance that the private key can exist only in a single cryptographic module. These retained keys are part of the sensitive data that are subject to zeroization.

2.2.6 Multiplatform support

The 4765 card is available to be plugged in to several IBM platforms:

- In the IBM System z, as packaged in the Crypto Express3 feature, with the software support in z/OS or Linux for System z.
- In the IBM Power Systems™, where the IBM 4765 PCIe adapter is available as the following features (all features correspond to a 4765 coprocessor but with a different packaging):
  - 4807
  - 4808
  - 4809

Software support is provided in the IBM AIX® operating system.

2.2.7 CCA platforms interoperability

The support of the 4765 card on multiple platforms is designed to strictly abide with the IBM CCA principles of operation and therefore guaranties CCA interoperability between these platforms.

2.2.8 Functional differences with the 4764-001

The main differences, mostly invisible from the host system, in the 4765 hardware implementation when compared to 4764-001, are:

- It uses a PCI-e interface to the host instead of a PCI Extended (PCIX) interface.
- Two PowerPC processors running CCA and Linux instead of one, for the sake of reliability.
- Two service processors, so that the code patch and loading process can run concurrently with CCA operations.
- The SHA-2 algorithm is supported, in addition to SHA-1.
- Two additional modular exponentiation engines, for a total of three engines instead of one.
- Faster processors, memory, busses, and so on.

The users are not to see functional differences at the CCA API externals level.

2.3 The System z Crypto Express3 feature

The System z Crypto Express3 feature can host one or two 4765 coprocessors:

- The Crypto Express3 feature with Feature Code 0864 hosts two coprocessors and can be installed either in System z10 EC or BC.
- The Crypto Express3-1P feature with Feature Code 0871 hosts only one coprocessor and can be installed in System z10 BC only.

The maximum number of coprocessors available on a single System z10 can therefore be 16 coprocessors when eight Feature Codes 0864 are installed.

Note that each coprocessor acts independently from the other coprocessors. Increasing the number of coprocessors potentially increases the global cryptographic throughput of the system. Increasing the number of coprocessors does not speed up the execution of individual cryptographic services.

2.3.1 Export control

Export of the Crypto Express3 functions is controlled with the LIC Feature Code 3863. If the feature code is not installed in the system, the coprocessors in the Crypto Express3 features cannot be put into the operational state.

2.3.2 Specific usage characteristics of the 4765 coprocessor in System z

In this section we discuss specific usage characteristics of the 4765 coprocessor in System z.

Physical packaging and plugging location

In this section we discuss physical packaging and the plugging location.

PCIe interface

The System z does not use PCIe busses. In the CryptoExpress3 feature, the 4765 cards plug onto a motherboard that includes an interface converter and controller between the card PCIe interface and the System z 10 internal self-timed interconnect (STI) bus.

I/O plugging location and its impact on performance

The Crypto Express3 feature plugs into the System z 10 I/O cage and can be operated from any processing unit in the system. This implementation implies using communications being driven along wires running between the PU books and the I/O cages, with information being staged in the system HSA.

This implementation inevitably induces delays and latency that might significantly affect the performance of some cryptographic operations as seen from the user API. Typically, this affects the short-duration symmetric algorithm-based operations. In 2.7, “Crypto Express3 performance” on page 46, we provide excerpts of the performance figures officially published by IBM. It is up to the users to check whether the operation that they want to be performed can be performed by the System z10 CPACF with acceptable security conditions. (The CPACF does not implement the master key concept.) If this is the case, the CPACF performs noticeably faster than the Crypto Express3.

Note, however, that asymmetric algorithm-based operations (which the Crypto Express3 is the only hardware unit to provide in System z) are long-duration operations that make the wiring-induced delays and latency appear negligible with respect to the total elapsed time that it takes the coprocessor to complete the operation.

Logical partitioning support

Logical partition *domains* are implemented by the 4765 firmware when operating in System z. Domains are dedicated zones in the BBRAM, within the coprocessor secure boundary, that hold master keys and retained keys pertaining to each assigned domain.

Master keys and retained keys are subject to domain zeroization when this function is requested by the operator or when it is automatically triggered by the coprocessor in case of tampering detection.

Reminder about master keys

In IBM CCA there are three master keys dedicated to protecting three families of program-owned cryptographic keys:

- The DES master key (DES-MK) is dedicated to the encryption of DES and Triple-DES secure keys. This is a double length (112 bits) Triple-DES key.
- The AES master key (AES-MK) is used for the encryption of AES secure keys. This is an AES 256-bit long key.
- The PKA master key (ASYM-MK) is used for the encryption of RSA secure private keys and for encryption of the integrity checking data of the ICSF *trusted block* entities. This is a triple-length (168 bits) Triple-DES key.

Access control to the coprocessor functions

When operating in the Crypto Express3, the 4765 functions are executed under the default role. Accesses to the functions by the default role are controlled by the setting of access control points in the coprocessor. Note that access control points can be switched between the enable and disable state from the TKE workstation only, and that the setting of access control points pertains to the cryptographic domain that was selected at the TKE workstation.

For non-TKE users, all access control points are enabled with the exception of DKYGENKY-DALL and DSG ZERO-PAD unrestricted hash length and PTR enhanced PIN security, which are disabled for all users and require a TKE workstation to be switched enable.

A complete list of the access control points that are used by the Crypto Express3 coprocessor is given in the *ICSF Application Programmer's Guide*, SA22-7522-13.

4765 coprocessor functions that are not exploited in System z

The following functions are not exploited when the coprocessor operates in the System z Crypto Express3 feature:

- The SHA-1, SHA-2, and MDC support when the coprocessor is exploited by z/OS applications. The host CCA API software drives these requests to the System z CPACF.
- The MD5 function: This function is provided by the host CCA API software.
- The coprocessor concurrent firmware update and load: Although this facility is supported in the coprocessor hardware, the relevant host support is not implemented in System z10.

Retained key use is not recommended with the Crypto Express3 feature

Although RSA retained keys are supported in the CEX3C coprocessor, their use in System z is not recommended by IBM.

Coprocessor and accelerator mode

These are the possible modes of operation for each 4765 coprocessor in a Crypto Express3 feature. Switching between the two modes is achieved by manual intervention at the System's HMC/SE.

- The coprocessor mode: This is the default mode of operations, where the full range of cryptographic functions is available to programs invoking the coprocessor, using secure keys whenever it is required.
- When the coprocessor has been manually switched to the accelerator mode, only operations that exploit the fastpath capability of the coprocessor are available. (See an explanation of the 4765 fast path in 2.2.2, "Hardware overview" on page 31.) These are RSA operations that use cleartext-only keys.

2.4 CCA services provided by the Crypto Express3

The 4765 coprocessors in the Crypto Express3 feature provide CCA services to cryptographic applications executing in the host system, in a functional infrastructure composed of the host system CCA API support software, the 4765 coprocessor CCA application software, and the 4765 coprocessor hardware and firmware-based cryptographic capabilities.

2.4.1 System z host system CCA software support

z/OS and Linux for System z come with a full CCA support:

- With the Integrated Cryptographic Service Facility (ICSF) component of z/OS
- With the zcrypt cryptographic device driver in Linux for System z

There is a subset of CCA functions provided in the z/VSE and z/TPF software support.

There is no CCA support in z/VM, in that the guest systems running in the virtual machines should provide their own CCA API software support for cryptographic service requests to be driven to the physical cryptographic coprocessors.

2.4.2 Cryptographic standards supported by System z CCA implementations

A list of the supported standards is given in the *z/OS ICSF Overview*, SA22-7519-13.

2.4.3 The CCA functions performed by the CEX3C coprocessors

This section broadly summarizes the CCA cryptographic functions that are provided by the coprocessors in the Crypto Express3 feature. A breakdown of CCA callable services support by hardware configuration is available as an appendix in the *z/OS ICSF Overview*, SA22-7519-13. The description of each individual CCA callable service in the *z/OS ICSF Application Programmer's Guide*, SA22-7522-13, also specifies what hardware facility is required to perform the function.

Data protection using symmetric algorithms

The functions below are performed using secure keys only:

- Data encryption and decryption
  - Using the DES or Triple-DES algorithm
  - Using the AES algorithm
- DES and AES key generation, importation, and distribution

  The key generation can be random generation or based on a derivation algorithm. The symmetric key can be exported in a format compatible with remote devices such as ATMs for an automated remote loading of the key.

Data integrity using symmetric algorithms

ICSF provides several methods to verify the integrity of transmitted messages and stored data such as the Message Authentication Code (MAC) and Modification Detection Code (MDC) callable services. The MAC services require the use of secure keys.

Asymmetric algorithms support

As of the writing of this book, the CEX3C coprocessor provides only RSA hardware support to System z users through the CCA API. There is, however, an internal implementation of the Diffie Hellman algorithm that is reserved for the coprocessor's own use when securely communicating with the TKE workstation.

Depending on the function invoked, the RSA secret private key is required to be presented in secure key form or in clear text form.

As of the writing of this book, the maximum RSA key length supported by the CEX3C feature is 4096 bits. Callable services for asymmetric algorithms include:

- Support of the RSA key pairs management
  - Importation of public-private key pairs in clear text and externally encrypted forms
  - Key pair generation
  - RSA key format translation from 4765 format to smart card format
- RSA Digital signature function

  The following services require the use of a secure RSA private key:
  - Generation, with selection of the signature formatting method.
  - Verification
- Symmetric key encryption function

The CEX3C coprocessor can proceed with the encryption of secure symmetric keys (DES, Triple-DES or AES) by an RSA public key, for secure export of these keys, or by an RSA private key when importing such a secure key.

A specific function is also implemented to assist the symmetric key wrapping or unwrapping during the SSL/TLS protocol handshake with the formatting of the resulting symmetric key block.

- Modular exponentiation

  This function is a specific form of the symmetric key encryption function that programs can invoke whenever such an exponentiation has to be applied against a data string. This can be used, for instance, for assisting software implementation of asymmetric algorithms other than RSA that also rely on modular exponentiation.

- Support of RSA retained keys

  Specifying RETAINED is an option when invoking and generating RSA key pair generation by the coprocessor. The key pair is generated in the secure physical boundary of the coprocessor, and the private key is never to leave this physical boundary. The retained private key, when needed, is designated by a label and retrieved by the coprocessor from its protected internal memory.

Financial services support

These services require to use secure keys.

- Personal identification number (PIN) processing services: Typically generation, verification and translation, with support for different formats of the PIN block
- Credit card verification value or card security code processing: Typically generation and verification, with support for different card types and algorithms
- Support for secure messaging as implemented in the Europay - Master Card-Visa (EMV) protocol
- Support for composing/decomposing data blocks in the Secure Electronic Transfer (SET) protocol

Other functions supported

The Crypto Express3 hardware also implements a high-quality and high-speed random number generator.

2.5 The Crypto Express3 TKE interface

As mentioned in 1.5.3, “The TKE workstation feature” on page 14, the TKE workstation is an optional priced feature that allows managing, from a central point of control, master keys and operational keys in Crypto Express3 coprocessors. The TKE creates a highly secure logical channel via TCP/IP communications between the workstation and the CEX3C coprocessors that it manages, which is used to exchange sensitive data over TCP/IP, such as the value of the master key to be set or coprocessor status information. The security of the logical channel relies on the use of strong encryption and a digital signature of the exchanged information.

This secure logical channel can be represented as shown in Figure 2-4, where encrypted and signed communications are still transiting via ICSF and are conveyed by CPRBs between ICSF and the 4765 coprocessor.

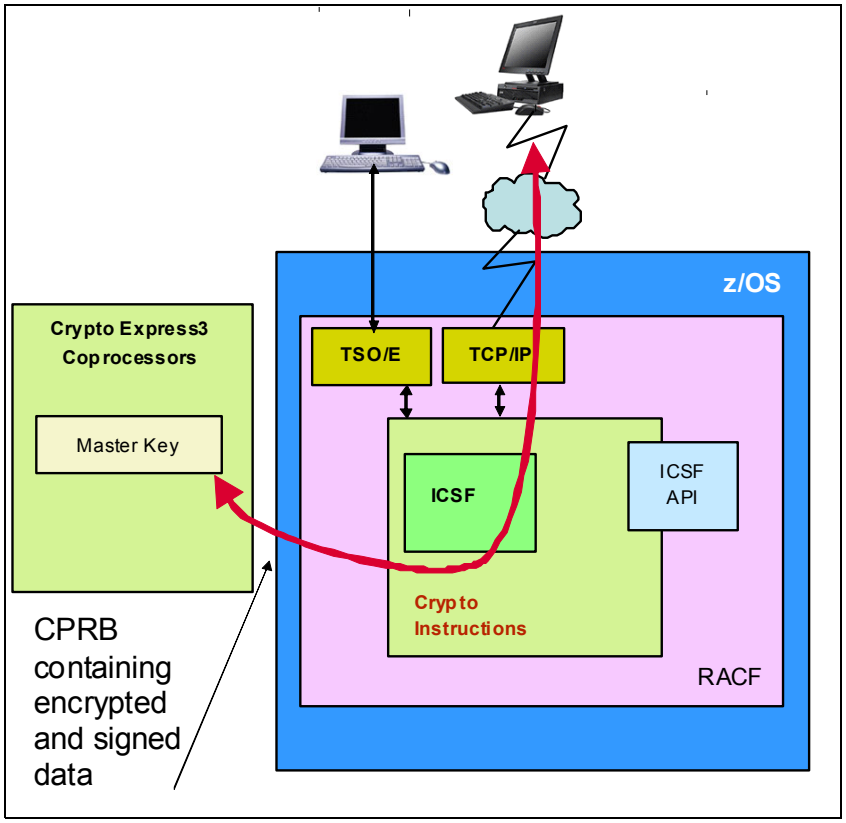


Figure 2-4 The Trusted Key Entry (TKE) workstation and The Crypto Express3 coprocessor

Digital signature of communications

The 4765 coprocessor owns a unique RSA key pair with a key length of 4096 bits. The private key is used to digitally sign the data that it sends to the TKE workstation. Likewise, the TKE workstation signs the data that it sends using a 2048-bit RSA key. This has to be compared with a the 1024-bit RSA key initially used with the 4764 coprocessor.

Encryption of sensitive data

Sensitive data to be exchanged between the 4765 coprocessor and the TKE are encrypted using an AES 256-bit transport key. The transport key is generated using the Diffie-Hellman key agreement protocol with a 2048-bit prime modulus and a 2048-bit full random exponent.

This has to be compared with the Triple-DES 168-bit key used with the 4764 coprocessor.

2.6 Introduction to the Crypto Express3 User Defined Extension (UDX) support

A UDX is a customer-designed extension to the 4765 CCA application software. As such, it is to be executed under the 4765 operating system and performs the cryptographic process that the customer designed against the data received from ICSF in the CPRB.

The actual cryptographic functions executed during this process can be coded in the UDX code itself, or the UDX code can invoke the native 4765 cryptographic and storage functions via a lower level API and can also invoke other services in the coprocessor CCA application software.

The source code of a UDX is written in C and processed by an IBM-developed toolkit that provides the required compilation libraries and debugging and packaging tools. Typically, a UDX is developed and tested on a workstation equipped with a coprocessor card and, when completed and packaged, eventually loaded into the customer's Crypto Express3 coprocessors. The UDX can be invoked only if the 4765 coprocessor operates in coprocessor mode.

Note: The UDX facility has been made available on the IBM mainframes with the PCI-based cryptographic coprocessors. The UDX development process is sensible to the coprocessor technology that is to execute the UDX.

Users who have UDXs developed for former PCI-based coprocessor technologies should find out from IBM what prerequisites and migration steps are required to make these UDXs executable in the Crypto Express3 4765 coprocessor.

2.6.1 The UDX development process

IBM has developed a UDX toolkit for the 4765 Cryptographic Coprocessor, which is part of a services offering on a custom contract. Such contracts normally provide education on preparing programs to operate within the coprocessor, a copy of the toolkit, follow-up support, and assignment of a unique identifier for the customer code and certification of the code-signing keys.

Availability of the toolkit is subject to the export regulations of the United States government, and in some cases the import regulations of other countries. At the present time, IBM is generally able to export the toolkit to customers within the European Union and to customers in Australia, Canada, Japan, and New Zealand.

Information about the IBM UDX offering and names of contacts can be found at:

http://www-03.ibm.com/security/cryptocards/

Note that as of the writing of this book there is not yet documentation available on the 4765 UDX development process. For nowreaders can refer to the 4764 Custom Programming library at:

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

Important: The output of the 4765 UDX generation process is a complete segment 3 load (that is, UDX *and* the regular CCA service modules, all together link-edited into a single load). As a consequence, each 4765 installed in System z might potentially have its own unique segment 3 code load.

2.6.2 UDX Installation

The UDX is delivered as a LIC CD to be loaded from the System z HMC or SE by performing the following steps:

1. The 4765 coprocessors targeted for UDX loading are deactivated at the ICSF coprocessors status panel.
2. The UDX file on diskette (the custom segment 3 load) is imported via the HMC or the system support element panels.
3. The UDX code load is activated and is loaded into one or more 4765s.
4. The target 4765s are activated again at the ICSF panel.

This can be done concurrently with the system's operations.

At any time the customer can elect to reload the original IBM segment 3 using the "Reset UDX to IBM image" function at the HMC or SE.

2.6.3 The UDX functional implementation

There are three elements of the UDX functional implementation:

- An ICSF callable service, which when called by a user application invokes the UDX code to be executed in the 4765 coprocessor. This is a user-defined host service that matches the specific UDX code running in the coprocessor card. Applications call the service using a designated service number.
- The 4765 UDX code. This is the coprocessor piece installed by the user as an extension to the 4765 CCA support application in segment 3 and is intended to be invoked by the ICSF service above. A UDX is designated by a specific two-character identifier.
- A service *stub* to be link-edited with the application calling the user-defined service for proper dispatching of the service by ICSF.

Additionally, the specific UDX service can be enabled or disabled using access control points managed from the TKE workstation. This requires you to design and install an exit to the CSFPCI (TKE communication) ICSF callable service.

Details on the process steps that involve ICSF are given in the *z/OS ICSF System Programmer's Guide,* SA22-7522-13.

The 4765 code structure and UDX

As discussed in 2.2.3, "Coprocessor software layout" on page 34, the 4765 coprocessor code is organized by segments in the card's flash memory. Segment 3 contains the code executing the CCA services and is implemented as a set of CCA command processors.

The UDX code to be installed in a 4765 consists of a user-designed command processor that is link-edited into the segment 3 load module during the UDX generation process. Therefore, creating a UDX requires you to acquire the proper tools for designing, debugging, and packaging a new segment 3 load. As for the base segment 3 load, a segment 3 with UDX load is checked for origin and integrity using a digital signature scheme. The new segment 3 load, with the UDX integrated, should have been signed by an IBM approved RSA key.

Note that a customer can elect to have more than one UDX in a single coprocessor. Each UDX will be uniquely referred to using its function identifier (refer to "ICSF and 4765 coprocessor communications" on page 45).

Segment 2 contains the Linux operating system that interfaces with ICSF for getting the function requests and issuing the related responses. When receiving a request for cryptographic function, either from ICSF or internally from an executing command processor, the operating system controls the dispatch and execution of the command processor providing the requested CCA or UDX function. An internal table containing all the entry points for the CCA and installed UDX command processors is used. This table is updated following a coprocessor reset sequence.

Included in the controls performed prior to dispatching a command processor is also the permission given, via access control point, to the 4765 default role to use the targeted command processor.

ICSF and 4765 coprocessor communications

As seen in 2.2.1, “Coprocessor principles of operation” on page 30, the z/OS ICSF requests to the ICSF selected coprocessor are conveyed using the Cooperative Processing Request/Response Block (CPRB). A CPRB contains a sub-function code, a two-character value, which is used to uniquely identify the required command processor. After parsing the received CPRB, the 4765 operating system uses the sub-function identifier to invoke the proper command processor. A UDX will have its own sub-function identifier.

2.6.4 Details on the UDX implementation

In this section we discuss the UDX implementation.

The UDX function code identifier

The UDX is in essence another command processor, as the CCA command processor, which has its own function identifier that is specified during the UDX generation process. The function identifier corresponding to an installed UDX is known to the 4765 operating system consequently to a coprocessor reset. However, the ICSF code shipped with z/OS cannot integrate in advance of the function code that a user might choose for her UDX. This is solved by adding in the options data set, once the UDX has been installed in the 4765, a UDX statement like the one that follows:

```
UDX(UDX-id,service-number,load-module-name,'comment_text',FAIL(failoption))
```

Where *UDX-id* is the function code that identifies the UDX command processor, *service-number* is to be used to refer to this CCA services extension, and l*oad-module-name* is the name of the user-defined callable service that is eventually calling the UDX in the 4765 (refer to the *ICSF System Programmer’s Guide*, SA22 7520, for a complete description of the statement).

UDX callable service and the stub

A UDX, not being a base CCA verb, requires a specific routine in ICSF to provide the service at the ICSF CCA API level. This ICSF callable service in turn manages to call the UDX in the 4765 coprocessor.

The capability of creating customized services in ICSF (the installation-defined services) has been available since the early releases of ICSF. These services are customer-written modules link-edited with ICSF and identified in the options data set (refer to *ICSF System Programmer's Guide*, SA22 7520) by the SERVICE statement where a unique number (from 1 to 32767, inclusive) is associated with the service load module.

After the callable service is written, it has to be link-edited into a load module, and the load module installed into an APF authorized library. ICSF uses the following normal search order to locate the service:

- Job pack area
- Steplib (if one exists)
- Link pack area (LPA)
- Link list (SYS1.LINKLIB concatenation)

During ICSF startup, ICSF loads the load module that contains the service into the ICSF address space with the other ICSF callable services. ICSF binds the service with the service number that is specified in the installation options data set. To call such an installation-defined service the application has to call an intermediary piece of code called the *stub*. The service stub is also designed and installed by the user and must do the following:

- Check that ICSF is active.
- Place the service number for the installation-defined callable service into register 0.
- Call the IBM-supplied processing routine, CSFAPRPC, which is internally used by ICSF to access the callable services. This tells the stub to retrieve first the location of CSFAPRPC from the ICSF cryptographic communication vector table (CCVT).

A stub example is given in the *ICSF System Programmer's Guide*, SA22 7520.

Any application program that calls a service stub must be link-edited with the service stub. The following CALL statement has to be used to call an installation-defined service from an application program:

```
CALL <service-stub-name><service-parameters>
```

service-stub-name is the name of the service stub for the installation-defined callable service. The *service-parameters* are the parameters to be passed to the installation-defined service. The parameters are supplied according to the syntax of the programming language used to write the application program.

2.7 Crypto Express3 performance

IBM officially publishes a set of measurements pertaining to the exploitation of hardware cryptography in System z. These measurements are performed using test programs and can be accessed at:

http://www.ibm.com/servers/eserver/zseries/security/cryptography.html

This section shows measurements data that help position the Crypto Express3 performance with respect to the CPACF and Crypto Express2 one, when they are called by z/OS applications.

Note: The published measurements documents do not directly provide a Crypto Express3 versus Crypto Express2 comparison. For the sake of providing such a comparison, with an acceptable approximation range, the Crypto Express3 figures are excerpts of the System z10 GA3 measurements, whereas the Crypto Express2 figures come from the same types of measurements previously done on a System z10 GA1.

2.7.1 Symmetric encryption performance: Crypto Express3 versus CPACF and Crypto Express2

Figure 2-5 is an example of a throughput comparison between one CPACF and one Crypto Express3 feature when encrypting data using the Triple DES and AES algorithm. The Crypto Express3 and Crypto Express2 proceed with encryption using secure keys, while the CPACF uses a clear key or a protected key.

As indicated in Figure 2-5, the data are provided for encryption by blocks of 1024 bytes, and there is only one thread of requests, thus using only one coprocessor in the Crypto Express3 feature. The published measurements also show the throughput obtained when seven jobs are concurrently submitting requests. In that case the Crypto Express3 throughput observed for one job is increased by a factor of 1.5 to 2.4.

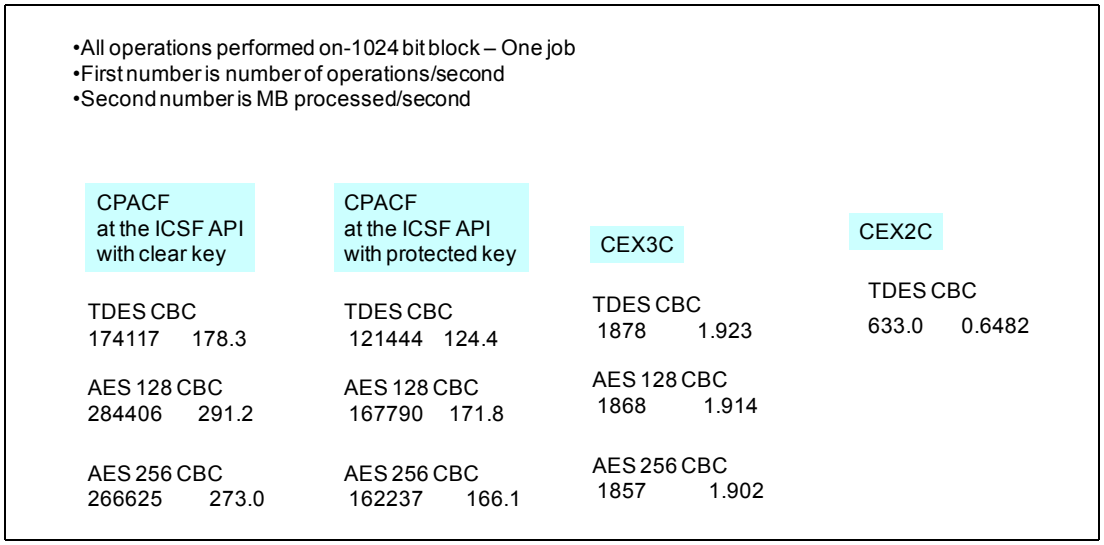


Figure 2-5 CPACF symmetric encryption versus Crypto Express3: one job

2.7.2 Asymmetric encryption performance: Crypto Express3 versus Crypto Express2 in coprocessor mode

Figure 2-6 is an example of measurements performed using the PKA Decrypt callable service (PKD, which is encryption using an RSA private key) by Crypto Express3 and Crypto Express2 coprocessors running in coprocessor mode. The private key used is a cleartext private key, with the length, in bits, of the key indicated with the measurements results.

The measurements are done varying the number of CEX3C 4765 coprocessors in action and the amount of jobs concurrently submitting requests.

	2097 E26	2097 E26	2097 E26	2097 E26
CEX2C	1	1	2	4
Jobs	1	7	14	28
	Operations/sec	Operations/sec	Operations/sec	Operations/sec
PKD--CRT, 1024 bit	631	1130	2261	4525
PKD--CRT, 2048 bit	273	466	932	1861
PKD--CRT, 4096 bit	41	44	88	175

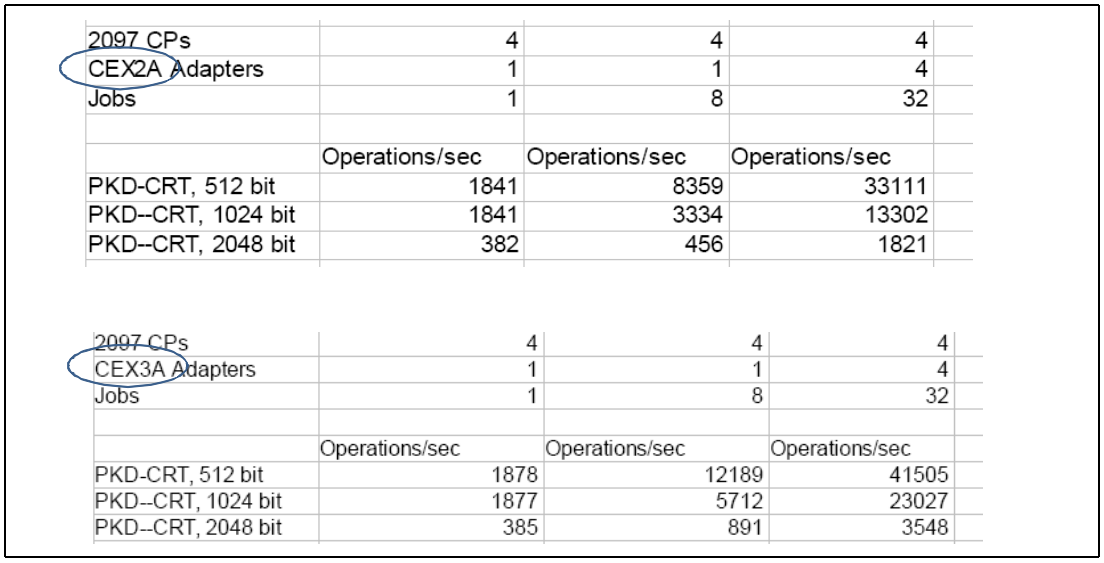
	2097-754	2097-754	2097-754	2097-754
CEX3C	1	1	2	4
Jobs	1	7	14	28
	Operations/sec	Operations/sec	Operations/sec	Operations/sec
PKD--CRT, 1024 bit	953	2080	4157	8162
PKD--CRT, 2048 bit	321	932	1862	3715
PKD--CRT, 4096 bit	45	88	176	350

Figure 2-6 Asymmetric encryption: Crypto Express3 versus Crypto Express2: Coprocessor mode

2.7.3 Asymmetric encryption performance: Crypto Express3 versus Crypto Express2 in accelerator mode

Figure 2-6 on page 48 is an example of measurements performed using the PKA Decrypt callable service (PKD, which is encryption using an RSA private key) by Crypto Express3 and Crypto Express2 coprocessors running in accelerator mode. The private key used is a cleartext private key, with the length in bits of the key indicated with the measurements results.

The measurements are done by varying the number of CEX3C 4765 accelerators in action and the amount of jobs concurrently submitting requests.



2097 CPs	4	4	4
CEX2A Adapters	1	1	4
Jobs	1	8	32
	Operations/sec	Operations/sec	Operations/sec
PKD-CRT, 512 bit	1841	8359	33111
PKD--CRT, 1024 bit	1841	3334	13302
PKD--CRT, 2048 bit	382	456	1821

2097 CPs	4	4	4
CEX3A Adapters	1	1	4
Jobs	1	8	32
	Operations/sec	Operations/sec	Operations/sec
PKD-CRT, 512 bit	1878	12189	41505
PKD--CRT, 1024 bit	1877	5712	23027
PKD--CRT, 2048 bit	385	891	3548

Figure 2-7 Asymmetric encryption: Crypto Express3 versus Crypto Express2 - Accelerator mode

3

CP Assist for Cryptographic Functions (CPACF)

This chapter describes the CPACF facility as provided on the System z10 processors. We include a history of CPACF, showing the growth of function within the facility, and then we describe the design and use of protected keys with CPACF.

Examples are given of how to use protected keys with and without a Crypto Express3 device.

3.1 Introduction to CPACF

CP Assist for Cryptographic Functions was first introduced on the z990 and z890. CPACF provides various cryptographic functions, but until recently the encryption and decryption functions worked only with clear keys.

CPACF operates with a specific set of machine instructions, the Message-Security Assist (MSA) instructions, which are predominantly problem-state instructions and therefore available to all applications. (There is a single instruction that is deemed a control instruction and that requires the processor to be in supervisor state to execute.) The MSA instructions are described in *z/Architecture Principles of Operation*, SA22-7832.

CPACF does not supply instructions for performing asymmetric encryption.

The MSA instructions are all executed synchronously with respect to the CP instruction stream, unlike operations executed on the Crypto Express cards, which execute asynchronously. CPACF operations are therefore fast and can be used to support a high volume of cryptographic requests. CPACF instructions are available on every PU within System z (including IFLs, zIIP, and zAAP processors).

Also, unlike the instructions for the Crypto Express devices, the MSA instructions are invoked as machine instructions within an instruction stream (that is, a program). The Crypto Express devices operate asynchronously and require the ICSF to manage them.

Also, unlike the Crypto Express features, there is no concept of logical partition sharing or cryptographic domains with CPACF.

Until the System z10 processor with GA3 microcode, the CPACF operated with clear keys only. However, the z10 processor with GA3 microcode supports a new type of key processing called protected key processing. (This is occasionally referred to as *high performance secure key processing*, but that is not a preferred term.) We examine protected keys in detail.

3.1.1 Enablement

To make use of most of the CPACF functions it is necessary to install Feature Code 3863. If this is installed, then this can be viewed from the HMC or Support Element.

Using an HMC or the IBM System z9 or z10 Support Element, you can verify that the processor has the feature installed, as follows:

1. From the Views area, open **Groups and CPC**.
2. Select the CPC icon in the CPC Work Area view and double-click to open the CPC details window.
3. In the window, verify that the CPACF enablement feature Feature Code 3863 is installed.

Some of the CPACF MSA instructions do not require enablement. These are the SHA-1 and SHA-256 functions.

On the System z10 GA3 processor, further controls are available over the use of the PCKMO instruction. This can be separately enabled or disabled for either DEA or AES operations within the LPAR profile. It can also be enabled and disabled in flight. The default setting is for the instruction to be enabled. See Figure 3-1 for the LPAR view of these settings on the HMC.

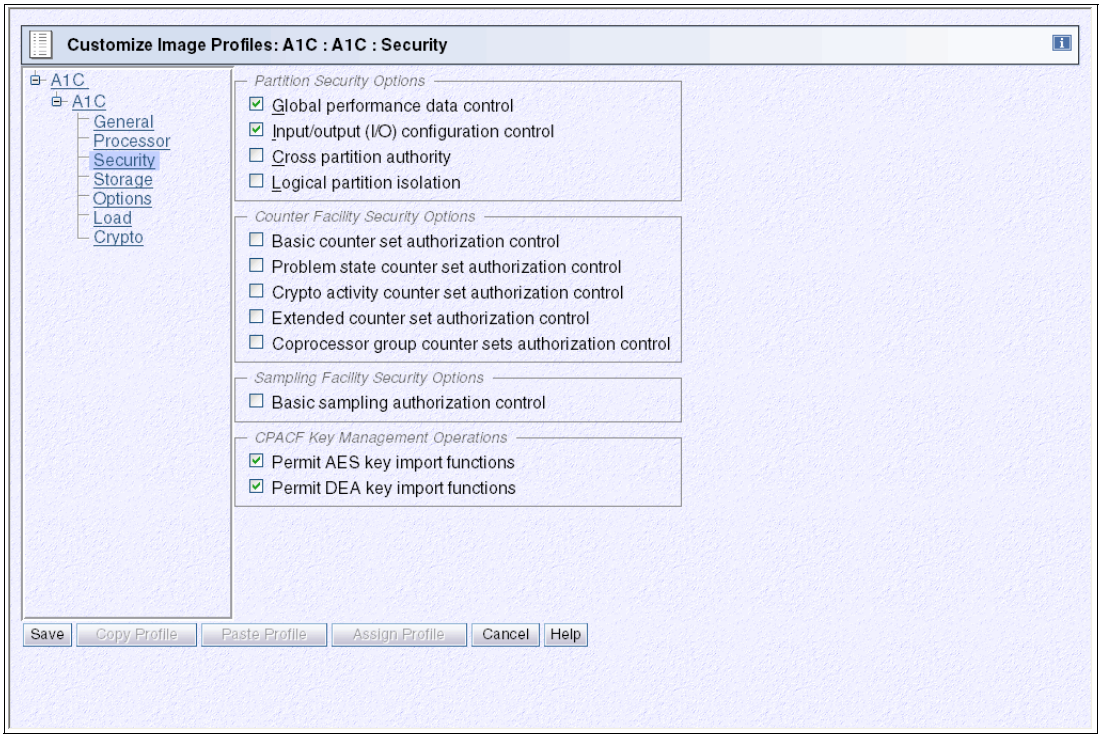


Figure 3-1 LPAR view

3.1.2 What the Message-Security Assist instructions do

The MSA instructions provide for four types of cryptographic operations. These are:

- Symmetric Encryption and Decryption (clear key only)
- Generation of hash values
- Generation of Message Authentication Codes (MAC)
- Pseudo random number generation

In addition, each function supplies a query option so that the programmer can determine whether a given function is available on a given processor. Attempted use of a function that is not available yields a program interruption with interruption code 6 (specification exception). In z/OS this is normally presented as an 0C6 abend.

Lastly, there is a new CPACF instruction for the z10 GA3 microcode level, called Perform Cryptographic Key Management Operation. We look at that instruction when we examine the use of protected keys.

We next examine each of the main MSA functions in turn.

Symmetric encryption and decryption

This is one of the most basic functions and involves the use of a key that is used to encrypt a string of bytes using an algorithm. This encryption mechanism is *symmetric* because the same key is used to encrypt and decrypt.

The byte string might be very long, especially in the case of the 64-bit version of the instruction. The key can be of several different types:

- DES keys: These can be single length, double length, or triple length.
- AES keys: These can be 128 bit, 192 bit, or 256 bit. The support level differs with regard to the System z processor. See Table 3-1 on page 55 for further details.

Note that DES is an abbreviation for Data Encryption Standard (DES) and that the instructions provide processing equivalence to the Data Encryption Algorithm (DEA), which is used at the heart of DES. Broadly speaking, you can consider these to be equivalent. The same equivalence applies to Triple DES (TDES) and TDEA.

The Advanced Encryption Standard (AES) is sometimes known as Rijndael. It is a block cipher adopted as an encryption standard by the U.S. government. It is considered the successor to DES and TDES and is now used worldwide. AES was adopted by the National Institute of Standards and Technology (NIST) as US FIPS PUB 197 in November 2001 after a 5-year standardization process.

Generation of hash values

Hash values are generated against data strings using a specified and repeatable algorithm. No key is ever involved in the generation process. Hash values are used for various functions, including:

- Mapping a large number of items to a smaller naming space.
- Providing a form of a check pattern to detect changes to a piece of data. In this case it is sometimes known as a *message digest*.

Due to the method used, and the size of the digest produced, it is theoretically possible that several different data configurations will yield the same message digest value. Such a possibility is called a *collision*. It is expected that hash algorithms will keep the risk of yielding collisions as low as possible, and will keep voluntary discovery of data configurations that would result in collisions as difficult as possible.

A commonly used hash algorithm today is Secure Hash Algorithm-1 (SHA-1), which generates a 20-byte hash value.

SHA-256 is an improved algorithm and generates a 32-byte hash value. SHA-256 is considered to generate message digest values that are less likely to yield collisions.

Hashing techniques are frequently used for message integrity checking. A hash value can be generated for a given message and it can then be encrypted using an asymmetric encryption technique. This can then be used to confirm that a message has not been altered en route and also to prove proper origin of the message. This technique is called *digital signature*.

Generation of message authentication codes

Messages can be authenticated using message authentication code (MAC) values. MAC values are generated using a block of data and a secret symmetric key as input. Thus, MAC codes can be used to confirm that messages have not been altered during transmission and are from the claimed origin. To do this, the sender and the receiver must both know the secret key used to produce the MAC. This same key can then be used to verify the MAC. If the message has been changed in any way, then the MAC value will no longer show a match between the one received and the one generated at the reception.

The objective of the MAC is similar to the digital signature. However, it relies on symmetric encryption, which is more efficient than asymmetric encryption, but can involve more complex key distribution schemes because the key to be used by both parties is a secret key.

Pseudo random number generation

Random numbers are very important for cryptographic applications, and it is important that the randomness be of good “quality”. The CPACF pseudo random number generator (PRNG) uses a variation of TDEA. The algorithm is described in *z/Architecture Principles of Operation,* SA22-7832-08. Note that use of the PRNG function still requires a random seed.

3.2 Calling the CPACF via ICSF services

Certain ICSF services make use of the CPACF instructions. This is sometimes a useful way of invoking the CPACF functions, and it is supported from high-level languages as well as from assembler.

The following ICSF services invoke CPACF hardware functions:

- Symmetric Key Encipher (CSNBSYE) and Symmetric Key Decipher (CSNBSYD), with a keyword of AES or DES in the rule array
- One-Way Hash Generate (CSNBOWH), with the SHA-256 keyword in the rule array.

There is no ICSF service for the following CPACF functions:

- MAC: ICSF provides the MAC services (MAC Generate, CSNBMGN, or MAC Verify, CSNBMVR) only if at least one Crypto Express Coprocessor is available in the system.
- PRNG: ICSF does not use the CPACF Pseudo Random Number Generator. ICSF provides the Random Number Generate service (CSNBRNG) only if at least one Crypto Express coprocessor is available in the stem.

Note: Invoking the CPACF functions via ICSF adds instruction path length to the execution of the involved cryptographic services. However, application developers can benefit from some additional functions provided to ICSF, like RMF reporting for SHA activities and the capability of using clear keys stored in the CKDS, along with RACF protection of the keys.

3.3 History of CPACF

The CPACF was introduced first on the z990 processor and then on the smaller z890 processor. The z9-EC (previously z9-109) and z9-BC processors had enhanced facilities, including the first AES algorithm support. The z10-EC and z10-BC processors then introduced support for longer AES keys.

The z10 GA3 macrocode level introduced the support for protected key processing. Table 3-1 shows all the available CPACF functions available from the z990 through to the z10 with GA3 microcode.

Table 3-1 Message-Security Assist and CPACF functions - z990 and later

Instruction	Description	Explanation	Models available
KM - Query	Cipher message	Query available functions.	All: Updated results on z9 and later models
KM - DEA	Cipher message	Provide Electronic Code Book (ECB) encryption and decryption of data using a DEA algorithm with single length keys (8 bytes).	All

Instruction	Description	Explanation	Models available
KM - TDEA-128	Cipher message	Provide ECB encryption and decryption of data using a DEA algorithm with double length keys (16 bytes).	All
KM - TDEA-192	Cipher message	Provide ECB encryption and decryption of data using a DEA algorithm with triple length keys (24 bytes).	All
KM - Encrypted - DEA	Cipher message	Provide Electronic Code Book (ECB) encryption and decryption of data using a DEA algorithm with single length protected keys (8 bytes).	z10 - GA3 only
KM - Encrypted - TDEA-128	Cipher message	Provide ECB encryption and decryption of data using a DEA algorithm with double length protected keys (16 bytes).	z10 - GA3 only
KM - Encrypted - TDEA-192	Cipher message	Provide ECB encryption and decryption of data using a DEA algorithm with triple length protected keys (24 bytes).	z10 - GA3 only
KM - AES-128	Cipher message	Provide ECB encryption and decryption of data using an AES algorithm with 16-byte keys.	z9 and later
KM - AES - 192	Cipher message	Provide ECB encryption and decryption of data using an AES algorithm with 24-byte keys.	z10 and later
KM - AES - 256	Cipher message	Provide ECB encryption and decryption of data using an AES algorithm with 32-byte keys.	z10 and later
KM - Encrypted - AES-128	Cipher message	Provide ECB encryption and decryption of data using an AES algorithm with 16-byte protected keys.	z10 - GA3 only
KM - Encrypted - AES - 192	Cipher message	Provide ECB encryption and decryption of data using an AES algorithm with 24-byte protected keys.	z10 - GA3 only
KM - Encrypted - AES - 256	Cipher message	Provide ECB encryption and decryption of data using an AES algorithm with 32-byte protected keys.	z10 - GA3 only
KMC - Query	Cipher message with chaining	Query available functions.	All - Updated results on z9 and later models
KMC - DEA	Cipher message with chaining	Provide Cipher Block Chaining (CBC) encryption and decryption of data using a DEA algorithm with single length keys (8 bytes).	All
KMC - TDEA-128	Cipher message with chaining	Provide CBC encryption and decryption of data using a DEA algorithm with double length keys (16 bytes).	All
KMC - TDEA-192	Cipher message with chaining	Provide CBC encryption and decryption of data using a DEA algorithm with triple length keys (24 bytes).	All
KMC - Encrypted - DEA	Cipher message with chaining	Provide Cipher Block Chaining (CBC) encryption and decryption of data using a DEA algorithm with single length protected keys (8 bytes).	z10 - GA3 only

Instruction	Description	Explanation	Models available
KMC - Encrypted - TDEA-128	Cipher message with chaining	Provide CBC encryption and decryption of data using a DEA algorithm with double length protected keys (16 bytes).	z10 - GA3 only
KMC - Encrypted - TDEA-192	Cipher message with chaining	Provide CBC encryption and decryption of data using a DEA algorithm with triple length protected keys (24 bytes).	z10 - GA3 only
KMC - AES-128	Cipher message with chaining	Provide CBC encryption and decryption of data using an AES algorithm with 16-byte keys.	z9 and later
KMC - AES - 192	Cipher message with chaining	Provide ECB encryption and decryption of data using an AES algorithm with 24-byte keys.	z10 and later
KMC - AES - 256	Cipher message with chaining	Provide ECB encryption and decryption of data using an AES algorithm with 32-byte keys.	z10 and later
KMC - Encrypted - AES-128	Cipher message with chaining	Provide CBC encryption and decryption of data using an AES algorithm with 16-byte keys.	z10 - GA3 only
KMC - Encrypted - AES - 192	Cipher message with chaining	Provide ECB encryption and decryption of data using an AES algorithm with 24-byte keys.	z10 - GA3 only
KMC - Encrypted - AES - 256	Cipher message with chaining	Provide ECB encryption and decryption of data using an AES algorithm with 32-byte keys.	z10 - GA3 only
KMC - PRNG	Cipher message with chaining	Pseudo random number generation.	z9 and later
KIMD - Query	Compute intermediate message digest	Query available functions.	All - Updated results on z9 and later models
KIMD - SHA-1	Compute intermediate message digest	Provide SHA-1 generation for blocks of data in multiples of 64 bytes. (Note: This instruction does not require Feature Code 3863 to be enabled.)	All
KIMD - SHA-256	Compute intermediate message digest	Provide SHA-256 generation for blocks of data in multiples of 64 bytes. (Note: This instruction does not require Feature Code 3863 to be enabled.)	z9 and later
KIMD - SHA-512	Compute intermediate message digest	Provide SHA-512 generation for blocks of data in multiples of 128 bytes. (Note: This instruction does not require Feature Code 3863 to be enabled.)	z10 and later
KLMD - Query	Compute last message digest	Query available functions.	All - Updated results on z9 and later models
KLMD - SHA-1	Compute last message digest	Provide SHA-1 generation for blocks of data that are not multiples of 64 bytes. (Note: This instruction does not require Feature Code 3863 to be enabled.)	All
KLMD - SHA-256	Compute last message digest	Provide SHA-256 generation for blocks of data that are not multiples of 64 bytes. (Note: This instruction does not require Feature Code 3863 to be enabled.)	z9 and later

Instruction	Description	Explanation	Models available
KLMD - SHA-512	Compute last message digest	Provide SHA-512 generation for blocks of data that are not multiples of 128 bytes. (Note: This instruction does not require Feature Code 3863 to be enabled.)	z10 and later
KMAC - Query	Compute message authentication code	Query available functions.	All
KMAC - DEA	Compute message authentication code	Provide Message Authentication Code (MAC) generation using a DEA algorithm with single length keys (8 bytes).	All
KMAC - TDEA-128	Compute message authentication code	Provide MAC generation using a DEA algorithm with double length keys (16 bytes).	All
KMAC - TDEA-192	Compute message authentication code	Provide MAC generation using a DEA algorithm with double length keys (24 bytes).	All
KMAC - Encrypted - DEA	Compute message authentication code	Provide Message Authentication Code (MAC) generation using a DEA algorithm with single length protected keys (8 bytes).	z10 - GA3 only
KMAC - Encrypted - TDEA-128	Compute message authentication code	Provide MAC generation using a DEA algorithm with double length protected keys (16 bytes).	z10 - GA3 only
KMAC - Encrypted - TDEA-192	Compute message authentication code	Provide MAC generation using a DEA algorithm with double length protected keys (24 bytes).	z10 - GA3 only
PCKMO - Query	Perform cryptographic key management operation	Query available functions. Supervisor state required.	z10 - GA3 only
PCKMO - Encrypt DEA - Key	Perform cryptographic key management operation	Wrap a DEA key (8 bytes) using the DEA wrapping key and provide the DEA wrapping key verification pattern. Supervisor state required.	z10 - GA3 only
PCKMO - Encrypt TDEA - 128 - Key	Perform cryptographic key management operation	Wrap a TDEA key (16 bytes) using the DEA wrapping key and provide the DEA wrapping key verification pattern. Supervisor state required.	z10 - GA3 only
PCKMO - Encrypt TDEA - 192 - Key	Perform cryptographic key management operation	Wrap a TDEA key (24 bytes) using the DEA wrapping key and provide the DEA wrapping key verification pattern. Supervisor state required.	z10 - GA3 only
PCKMO - Encrypt AES - 128 - Key	Perform cryptographic key management operation	Wrap an AES key (16 bytes) using the AES wrapping key and provide the AES wrapping key verification pattern. Supervisor state required.	z10 - GA3 only
PCKMO - Encrypt AES - 192 - Key	Perform cryptographic key management operation	Wrap an AES key (24 bytes) using the AES wrapping key and provide the AES wrapping key verification pattern. Supervisor state required.	z10 - GA3 only

Instruction	Description	Explanation	Models available
PCKMO - Encrypt AES - 256 - Key	Perform cryptographic key management operation	Wrap an AES key (32 bytes) using the AES wrapping key and provide the AES wrapping key verification pattern. Supervisor state required.	z10 - GA3 only

3.4 Protected key theory

This section describes background to key types and why it has become necessary to develop the protected key mechanisms available in the z10 GA3.

Distinguishing key types

Prior to the advent of protected keys, we distinguished keys as secure keys or clear keys. The introduction of this book introduced the concept of secure keys and clear keys. Let us provide an explanation of those and then move on to introduce protected keys.

Secure key

A secure key is a key whose native value is never exposed outside the bounds of a tamper-proof device.

The IBM devices, such as the CCF, PCICC, PCIXCC, CEX2C, and CEX3C are all certified to various FIPS standards as being tamper-proof. Each of these devices holds master keys that can be used to encrypt (or wrap) operational keys so that those operational keys can be manipulated outside of the tamper-proof device. The key encryption uses one of several master keys which are held within the tamper-proof device.

When an operational key is to be used, it is passed into the tamper-proof device together with the data on which it is to operate. Once within the device, the operational key can be unwrapped (decrypted) using the appropriate master key, and then it can be used to perform the cryptographic operation.

Operational keys can also be generated within the tamper-proof device and then passed out of the device only when they have been wrapped (encrypted) with the master key. Thus, for the entire life cycle of a key, it is never available in the clear outside the bounds of a tamper-proof device.

Secure keys are suitable for the long-term storage of encrypted data.

The processing of data using secure keys is subject to latency delays as the data and key are passed to the tamper-proof device, and then the results of the cryptographic operation are returned.

Clear key

A clear key does not need a tamper-proof device. It is designated as a clear key only because at the point when it is used in a cryptographic operation it is visible within the main storage of the computer system (and so is capable of being copied and used elsewhere). Exposing the value of the key means that the privacy of data encrypted under that key is compromised.

Clear keys are less appropriate for the encryption of data for long periods of time. They are, however, very suitable for data, which is essentially transient in nature. A good example of this is the encryption of the stream data in a short-lived SSL or TLS session. These sessions are used to encrypt data that flows over unsecured networks. In this case, the clear key is used for a short time and is then discarded. Once the encrypted data has completed its

passage over the unsecured network, that data is re-processed by the receiving computer system.

Why we need protected keys

Until the z10 GA3, CPACF provided only clear key operations that were very fast in nature. However, the encryption was not secure, as the key needed to be a clear key, and so was potentially accessible to programs, and could have also been exposed in system dumps.

Note: ICSF will manage clear keys if required, but in this case it is more important than ever that access to the symmetric key store (the CKDS) is secured from prying eyes, using logical controls, such as RACF.

The Crypto Express 2 and Crypto Express3 devices provide secure key processing, but at a cost. The processing times are far too long for certain types of operations.

The challenge was to supply a fast symmetric encryption capability that was also secure. The protected key is IBM's answer to this challenge.

Note: The series of System z processors prior to the z990 was called z900 and z800. These processors used a tamper-proof device called a CCF. This device was physically close to the main processor, and so did not suffer the latency of operations as experienced by the Crypto Express 2 and 3. Hence, it could perform secure operations at far higher speed than the Crypto Express 2 and 3, which we use today. However, the CCF was more limited in function, and more difficult to update with new functions. Consequently, it was not used in the later System z processors.

Protected keys

Secure key processing ensures that a key is never exposed outside of the bounds of the tamper-proof device in which it is processed.

Clear key processing allows a key to be visible within the storage of the computer system at some point in the key's lifetime.

In contrast to both of the above, a *protected* key is not visible within the storage of the computer system, and can never be addressed from a program running on z/OS or Linux within an LPAR. However, the protected key *is* available outside the bounds of the tamper-proof device.

To see how the keys can be used while not accessible to programs, we must consider the area of storage within the System z server known as the hardware storage area (HSA). This is an area of storage that is never directly addressable by programs. It is used for the storage of data relating to the entire processor, rather than individual operating systems. However, within System z there is millicode that can access this area in a highly controlled manner. Also, we can arrange for z/Architecture instructions to make implicit use of this storage area. In particular, CPACF instructions can access data in this area.

The z10 GA3 microcode introduces *wrapping keys*, which are created each time that an LPAR undergoes a System z clear/reset operation. This operation is normally performed each time that the z/OS system is IPLed. The wrapping keys are held in the HSA and are specific to each LPAR.

If we had a clear key that was wrapped using this wrapping key, we could manipulate it in storage without its clear key ever being known. In addition, if we could pass the wrapped key to a CPACF instruction, that instruction could access the wrapping key in the HSA and

unwrap the key so that it could be used in a cryptographic operation. The z10 GA3 microcode supplies this capability. Thus, we have all the parts necessary to perform encryption and decryption operations using keys that are only exposed in the storage of the computer in a wrapped form.

We have a final part to solve, however. How do we get the key to be wrapped? There are two solutions to this. The first involves the use of Crypto Express3 device and ICSF services. This solution is the preferred solution for many reasons.

However, if a Crypto Express3 is *not* available, then an alternate solution is possible using a new CPACF instruction called Perform Cryptographic Key Management Operation (PCKMO). This instruction is a privileged instruction and so requires the program to be in supervisor state in order to execute it.

In the next section we show how these methods work, but before we summarize what we have just discussed.

Summary

In summary:

- An operational key is used to encrypt data. An operational key can be a secure key, a protected key, or a clear key.
- A secure key is never exposed outside the bounds of a tamper-proof device.
- A clear key might be found within main storage by a programmer or someone reading a system dump.
- A master key is held in a tamper-proof device and is used to wrap an operational key so that it can be manipulated outside the bounds of a tamper-proof device.
- A wrapping key is generated each time that an LPAR undergoes a clear/reset operation (usually once per IPL).
- Wrapping keys are held in the HSA.
- HSA is not addressable by programs or by the operating system.
- A wrapping key can be used to wrap an operational key so that it can be passed to CPACF instructions in a more secure manner than clear keys.
- A key wrapped by this wrapping key is a protected key.
- A protected key is not available within the storage of the program in a clear form, and cannot be found by examination of a system dump.
- A protected key is stored outside of the bounds of a tamper-proof device (that is, in the HSA).

See Figure 3-2 on page 62 for a view of how the protected key wrapping works with the KM instruction.

3.5 How to use CPACF protected keys

This section describes how CPACF protected keys can be used to encrypt data and perform other cryptographic operations. There are two ways of achieving this. One method uses a secure key in the CKDS as the source key. The other method uses another key as a source. This needs to be supplied originally in clear form. The example that we use makes use of the CKDS again, and uses a CKDS clear key.

First we discuss the preferred mechanism using the Crypto Express3 feature.

3.5.1 Method 1: Using ICSF secure keys as protected keys

The most secure way of using protected keys is by using a Crypto Express3 device in conjunction with ICSF. Using this method requires the following components:

- z10 GA3
- ICSF level HCR7770
- Crypto Express3
- z/OS V1 R11 *or* z/OS V1 R10 with the PTF for APAR OA28439

This method provides for the transfer of a secure that which is stored in the ICSF CKDS to be converted into a protected key within an LPAR. Encryption and decryption can be handled via the ICSF calls CSNBSYE and CSNBSYD in a similar manner to clear key operations using the other CPACF instructions.

These two APIs have been updated (in ICSF level HCR7770) to support secure keys. For AES keys we have no control vectors, so this could potentially be any key. However, for DES keys this applies only to DATA keys. Keys with other control vectors cannot be used as protected keys.

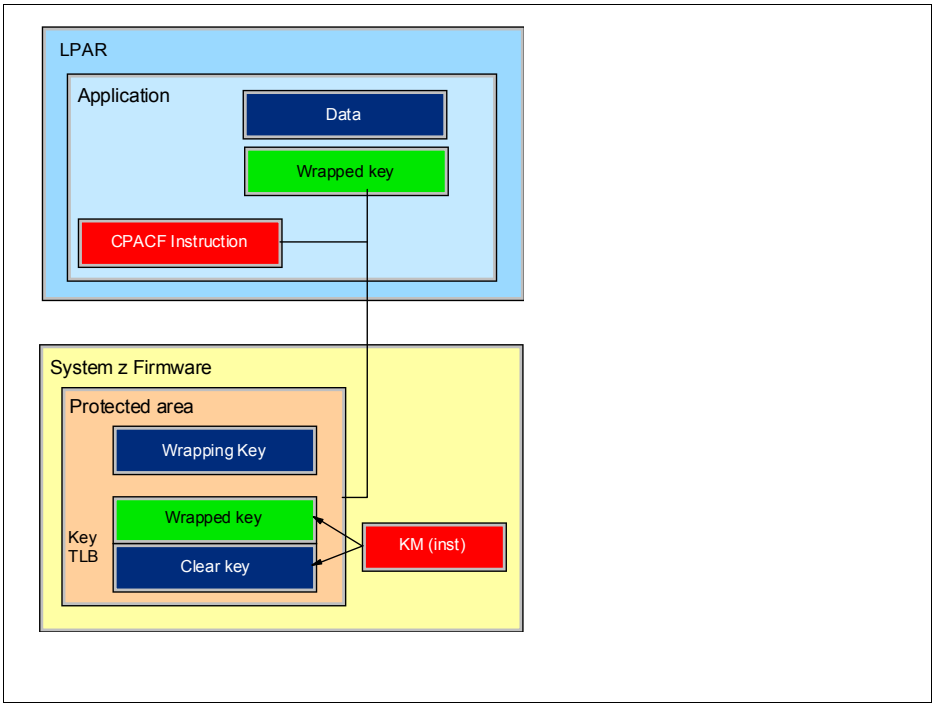


Figure 3-2 Diagram showing wrapped keys as used by a KM instruction

In this method the source key is within a secure token in the CKDS. The caller simply uses the ICSF service CSNBSYE to perform encryption.

Note: The caller has to specify the KEYIDENT option in the rule-array to show that he is using a token stored in the CKDS. The key_identifier is set to use the label of a secure key that previously has been defined.

ICSF recognizes that a secure key has been requested and checks that all the necessary prerequisites have been satisfied. We will come back to this later.

At this point some magic occurs “under the covers”. ICSF issues a specialised request to the Crypto Express3 feature, which arranges for the true value of the secure key to be passed to the System z Millicode, which then encrypts it (wraps it) using the LPAR wrapping key. It is then stored in the HSA, and the wrapped key and the wrapping key verification pattern are passed back to the caller. However, you might remember that the caller was ICSF, so ICSF now stores this information in a local cache in the ICSF address space and associates it with the key label passed to ICSF.

ICSF still has more to do, as the encryption operation has not yet been performed. However, at this point ICSF has the wrapped version of the secure key, together with the wrapping key verification pattern, and so can invoke the CPACF instruction KM (or KMC, as appropriate) to perform the encryption.

Finally, ICSF passes the encrypted data back to the caller.

You will notice that the caller does not need to perform any more work than he would have to if using a clear key.

How the ICSF cache is used

If a subsequent operation requests use of the same label, then the ICSF cache will already hold all the information necessary to specify the CPACF instruction needed to perform the operation. There is no need to repeat the “under the covers” operation of converting the secure key to a protected key.

For secure keys that have been requested to be made available as protected keys, ICSF will hold the label, the wrapped key version, and the key verification pattern.

The first use of a key will have a minor overhead. Subsequent uses will avoid this overhead and be able to move straight to the CPACF operation.

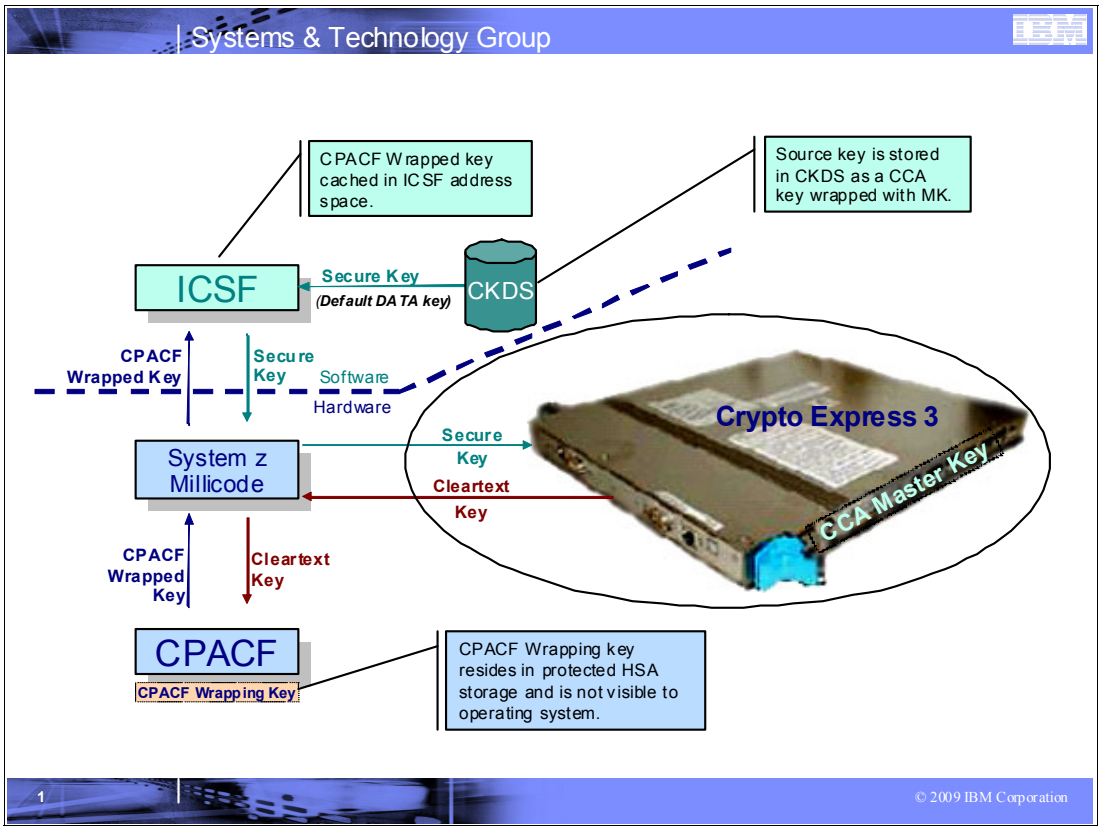


Figure 3-3 Overview of protected key processing using method 1

Advantages of this method

This method has some very nice features. For example, the coding necessary in the application is just as simple as it was when a clear key was used. The key label simply needs to point to a secure key rather than a clear key.

Note also that as the user has specified a label, all normal security checks can be performed against that label. So use of the secure key is not compromised in any way. The wrapped version of the key that is returned to ICSF is never made addressable to the caller, so all reference to the key is *still* via the label, thus ensuring the security of the key's use.

Note: There are certain situations in which the wrapping key can change (for example, if running under z/VM 6.1 and a live guest migration occurred). In this case, the CPACF instructions terminate with condition code 1. ICSF will recognise this and automatically redrive the conversion from secure key to protected key.

Preventing a secure key from being used as a protected key

You might have some very secure processes in which protected key processing is not deemed secure enough, and so you want to specifically prevent those keys from being used as protected keys. How can we achieve this?

We said above that there were prerequisites to be satisfied before ICSF took the step of converting a secure key to a protected key. Among these are checking the software levels, checking the processor on which we are running, and checking that a Crypto Express3 is available. There is another check as well. This makes use of RACF, and it is why we need either z/OS V1 R11 or z/OS V1 R10 with the PTF for APAR OA28439.

For a secure key to be capable of being used as a protected key it has to have a resource in the RACF CSFKEYS class that matches its label and that contains an ICSF segment. Furthermore, this ICSF segment must have the option SYMCPACFWRAP(YES) specified. The default for the SYMCPACFWRAP parameter is NO, so even if a segment is defined, the key could not be used as a protected key unless the value is explicitly changed to YES. For example, to allow the key LENNIES.SECURE.AESKEY01 as a protected key you could issue the following commands:

```
RDEFINE CSFKEYS LENNIES.SECURE.AESKEY* UACC(NONE) OWNER(LENNIE)
RALTER CSFKEYS LENNIES.SECURE.AESKEY* ICSF(SYMCPACFWRAP(YES))
```

If the CSFKEYS class is RACLISTED at your installation you would also need to issue:

```
SETROPTS RACLIST(CSFKEYS) REFRESH
```

Thus, only the keys that are specifically designated to be used as protected keys can be used as protected keys.

Illustrating method 1

We tested this method of using a SECURE key as the source for a PROTECTED key using a simple REXX program. This program uses the ICSF API CSNBSYE to encrypt some data and specifies the correct options to make use of a SECURE key as the base for a protected key.

The program then decrypts the resulting cipher text using the normal SECURE key API CSNBSAD. The program then confirms that the decrypted text is identical to the original text.

The source of this REXX program called REXPK001 can be found in Appendix A, “Sample programs in assembler and REXX” on page 261.

The output you receive will be similar to that shown in Figure 3-4.

```
----------------------------------------

Running CSNBSYE to perform AES Encryption
        (Protected key)

         Key = LENNIES.SECURE.AESKEY01
        Data = Bonjour matelot. Comment ca va. Tres bien merci.
 Data length = 48

CSNBSYE OK
Clear_Text  (part1)...C296959196A499409481A3859396A34B
Cipher_Text (part1)...D9000754D8FE5A79287AFE26D4027203

Clear_Text  (part2)...40C39694948595A340838140A5814B40
Cipher_Text (part2)...66A9C74CFD45D9B9FCBD0635F8647CD0

Clear_Text  (part3)...E39985A240828985954094859983894B
Cipher_Text (part3)...D7A878CE74EDDD3853CDD8AFEFEE57F5

CSNBSYE completed




----------------------------------------

Running CSNBSAD to perform AES Decryption
        (secure KEY)

CSNBSAD OK
Cipher_Text (part1)...D9000754D8FE5A79287AFE26D4027203
Clear_Text  (part1)...C296959196A499409481A3859396A34B

Cipher_Text (part2)...66A9C74CFD45D9B9FCBD0635F8647CD0
Clear_Text  (part2)...40C39694948595A340838140A5814B40

Cipher_Text (part3)...D7A878CE74EDDD3853CDD8AFEFEE57F5
Clear_Text  (part3)...E39985A240828985954094859983894B

Clear_Text = Bonjour matelot. Comment ca va. Tres bien merci.
Clear_Text length = 48

CSNBSAD completed

Data decrypted by CSNBSAD using clear key, exactly matches
original data encrypted by CSNBSYE using protected key.
```

Figure 3-4 Output

3.5.2 Method 2: Using ICSF clear keys as protected keys

This method of using protected keys does not involve the use of a Crypto Express3. However, it relies on the provision of a clear key, which is then converted to a protected key using a new CPACF machine instruction, PCKMO.

This method requires only z10 GA3. This level of z10 microcode provides the PCKMO instruction, which can be used to convert a key in clear form, to a wrapped key. The wrapped key can be manipulated wherever required, but should not be stored long term. This is because it is subject to change whenever the LPAR wrapping key is changed.

As you will see, there are some disadvantages with this method. However, it is a far less expensive solution, as the Crypto Express3 device is not required.

At the start of this process one must have a key. This key can be a single-length, double-length, or triple-length DES key or can be any of the AES keys supported (that is, 128 bit, 192 bit, or 256 bit). One place this could be obtained from is the ICSF CKDS. The CKDS can store clear keys, and these can be read using the ICSF API, CSNBKRR.

Once having obtained the clear key, the PCKMO instruction is used to create a wrapped version of it, which can then be used directly in the CPACF instructions such as KM and KMC.

You will see immediately the inherent weakness of this approach, as this key must be available in some kind of clear format within storage. However, with careful programming, the time that it is available in clear can be kept to an absolute minimum. Certainly, it need be in storage far less than a similar operation carried out using clear keys all the time. Thus, the risk of key exposure is much reduced.

So if we used a clear key stored in the CKDS the steps are:

1. Read the clear key token using CSNBKRR.
2. Extract the key from the token.
3. Convert the key to a wrapped form using PCKMO.
4. Remove the clear key from storage.
5. Use KM or KMC instructions to encrypt and decrypt data. (Repeat step 5 as necessary.)

Note that step 1 requires the caller of CSNBKRR to be in supervisor state or a system key if a clear token is to be read. Also note that PCKMO is a privileged instruction, so it also requires supervisor state. Thus, this method should not be undertaken lightly, and will require some very careful planning and programming.

If the KM or KMC instruction terminates with condition code 1, this indicates that a verification pattern mismatch has occurred. In this case it will be necessary to re-drive the conversion of the clear key to the protected key (that is, the wrapping process). So steps 1 to 4 in the list above will need to be repeated.

Illustrating method 2

This method took far more effort to produce samples, and required several assembler programs. The details follow.

Due to the need for some of the code to run with APF-authorization we were unable to invoke everything from REXX code. Consequently, a batch job is used to invoke an APF-authorized program that will:

- Read the name of a key label from a file. The key label references a key token for a CLEAR AES key stored in the CKDS.
- Use CSNBKRR to read the corresponding CLEAR key token into storage.
- Use the PCKMO instruction to make the key available as a protected key.
- Write the wrapped key and the verification pattern to a file.

This APF-authorized assembler program comprises two routines. The first (CPACF050) performs the I/O to the files. The second (CPACF040) performs the CSNBKRR call and uses the PCKMO instruction. Each program issues several diagnostic messages to show progress. The JCL for this batch job is CPACF001.

For our test this batch job needs to run only once. Once it has completed, the clear key from the CKDS has been wrapped and can be moved around in storage and used without exposing its true value. The JCL that we use writes the wrapped key and its verification pattern to a data set.

Once the wrapped key and its verification pattern are available in this data set, they can be used by a REXX program to perform encryption. However, to perform the encryption another assembler program had to be written to make use of the KMC instruction. This program is CPACF010. However, as this does not need to be APF-authorized, this program can be invoked from REXX, and we use the program REXCP050 to do this. REXCP050 performs the CPACF protected key encryption using CPACF010, and then uses the ICSF call CSNBSYD to decrypt the data, specifying the same CLEAR AES key that we passed to the batch job. The final decrypted string is compared to the original string to demonstrate that the values are the same.

Note: This set of code samples does not include any code to re-drive the production of the protected key, if the KMC instruction should terminate with condition code 1.

To summarize, we have:

- CPACF040: Assembler subroutine to perform CSNBKRR and PCKMO
- CPACF050: Assembler program to perform file I/O and call CPACF040
- CPACF001: JCL to invoke CPACF050
- CPACF010: Assembler subroutine to perform clear key and protected key encryption using CPACF instruction KMC
- REXCP050: REXX routine to read wrapped key and verification pattern, perform protected key encryption, and then call CSNBSYD to perform clear key decryption

All of these sample programs can be found in Appendix A, “Sample programs in assembler and REXX” on page 261. When you run the REXCP050 program you should receive output similar to that shown in Figure 3-5.

```
------------------------------------------------------

 Running CPACFO1O to perform AES Protected Key Encryption

        AES Key = 66A297A9C48571750655D734AB14FA47D76BEC35880BFA5776F4AC93F15CF2DB
    Ver Pattern = DBOCC8844A727C15EADB32140C36443E64D1192FBD3FB675AA4B2EDE4BOBEF1B
Data to encrypt = Bonjour matelot. Comment ca va. Tres bien merci.
Data to encrypt = C296959196A499409481A3859396A34B40C39694948595A340838140A5814B40E39985A240828985954094859983894B
 Length of data = 48

CPACFO1O OK

Clear_Text  (part1)...C296959196A499409481A3859396A34B
Cipher_Text (part1)...D9OOO754D8FE5A79287AFE26D4027203

Clear_Text  (part2)...4OC39694948595A340838140A5814B4O
Cipher_Text (part2)...66A9C74CFD45D9B9FCBDO635F8647CDO

Clear_Text  (part3)...E39985A240828985954094859983894B
Cipher_Text (part3)...D7A878CE74EDDD3853CDD8AFEFEE57F5

CPACFO1O completed


------------------------------------------

Running CSNBSYD to perform AES Decryption

Key = LENNIES.CLEAR.AESKEYO1

CSNBSYD OK

Cipher_Text (part1)...D9OOO754D8FE5A79287AFE26D4027203
Clear_Text  (part1)...C296959196A499409481A3859396A34B

Cipher_Text (part2)...66A9C74CFD45D9B9FCBDO635F8647CDO
Clear_Text  (part2)...4OC39694948595A340838140A5814B4O

Cipher_Text (part3)...D7A878CE74EDDD3853CDD8AFEFEE57F5
Clear_Text  (part3)...E39985A240828985954094859983894B

Clear_Text = Bonjour matelot. Comment ca va. Tres bien merci.
Clear_Text length = 48

CSNBSYD completed

Data decrypted by CSNBSYD using clear key, exactly matches
original data encrypted by CSNBSYE using protected key.
```

Figure 3-5 REXCP050 sample output

3.5.3 Comparison of methods

In this section we provide a comparison of the methods.

Ease of use

Method 1 is far easier to program, once the Crypto Express3 installed. The coding can be done in REXX routines if necessary. No special systems programming work is required, and results can be easily verified against similar cryptographic operations using the secure key and calls such as CSNBENC, CSNBDEC, CSNBSAE, and CSNBSAD.

Method 2 requires some systems programming effort, as assembler code has to be generated to invoke the PCKMO instruction. The code needs to be APF-authorized to allow it to move to supervisor state, both to invoke CSNBKRR against a clear key, and also to use the PCKMO instruction.

Security

There are two issues here. The first is that method 1 makes use of all the RACF protections available with regards to accessing the secure key. This includes checking access to use the key in the CSFKEYS class. This checking will be done for every ICSF call, even though the key is cached as a protected key within ICSF and the HSA. Method 2 will not provide this level of control. The sole ICSF call made will be to read the clear key token, and in normal operation this will happen only once.

The second issue is that method 2 will expose the clear key within program accessible-storage for a short while. Method 1 avoids this exposure entirely.

Performance

While we performed no formal test between these two methods, it appears that method 2 will be faster. There is no need to invoke CPACF encryption and decryption instructions through ICSF, so that overhead is lost. There will be fewer RACF checks as well.

Both methods will suffer a small overhead (probably only once) when the key has to be wrapped. In method 1 this will be part of an asynchronous operation, while with method 2 this will be synchronous with respect to the executing program.

The sample programs can be adapted to perform performance tests if required. There are also statistics available online, as is detailed in 3.6.6, “Performance” on page 71.

Cost

Method 2 avoids the purchase cost and ongoing maintenance cost of the Crypto Express3, but requires a far larger one-time development effort.

3.6 More details about CPACF protected keys

In this section we discuss CPACF protected keys further.

3.6.1 Types of keys and encryption

CPACF protected keys can be single-length, double-length, or triple-length DS keys or can be AES keys of lengths 128 bit, 192 bit, or 256 bit.

If coding in assembler, the choice of instructions for encryption and decryption (KM, KMC) provides the capability of performing either Electronic Code Book (ECB) encryption, Cipher Block Chaining (CBC) encryption, or Cipher Feedback Mode encryption.

If using ICSF to perform the encryption, all modes can also be used and all can be used with protected keys. However, ECB mode is generally deemed insecure nowadays and so should not be used.

3.6.2 Verification patterns

Each time that a key is wrapped with a wrapping key, whether by using PCKMO or by having ICSF do it, a verification pattern is supplied back to the caller. This pattern has to be supplied to the CPACF instructions when performing cryptographic operations. It is used to confirm that the LPAR wrapping key has not altered since the key was wrapped, and to ensure that the correct wrapping key is then used to unwrap the key to perform the cryptographic operation.

The DES verification pattern is 192 bytes. The AES verification pattern is 256 bytes.

When using method 1, the use of the verification pattern is entirely hidden from the user. It is managed entirely by ICSF, which keeps a local cache of items used for protected key operation. The cache contains the LABELs of the secure keys that have been wrapped, the wrapped key, and the verification patterns. However, the ICSF caller always passes the key LABEL and so is insulated from this processing.

3.6.3 CPACF instructions for protected keys

To make use of protected keys, the assembler programmer can use the following instructions:

- KM: Cipher message
- KMC: Cipher message with chaining
- KMAC: Compute Message Authentication Code

3.6.4 ICSF APIs for protected keys

The ICSF APIs that can be used with protected keys are:

- CSNBSYE: Symmetric Key Encipher
- CSNBSYD: Symmetric Key Decipher
- CSNBSMG: Symmetric MAC Generate
- CSNBSMV: Symmetric MAC Verify

The CSFIQF: ICSF Query Facility API will report protected key capabilities.

3.6.5 TLB in HSA

Within the HSA a table is held of the most recently unwrapped protected keys. This table is not available to any program. It is used to avoid the unwrapping of a protected key where it has been recently unwrapped. Do not confuse this table with the cache held within the ICSF address space of the keys, labels, and verification patterns.

3.6.6 Performance

Testing has been performed for each of the methods of doing CPACF encryption on both the z10-EC GA3 and the z10-BC GA3. Testing was performed both with and without the ICSF APIs. The results show a variety of message lengths between 64 bytes and 1 megabyte, and use various encryption key lengths and perform both DES and AES encryption.

In the best cases, the difference between using protected key and clear key is around 1%.

The full results can be found on the following web page by clicking the links to the PDF documents on the right-hand side:

http://www-03.ibm.com/systems/z/advantages/security/z10cryptography.html

3.6.7 Key storing

Because the wrapping key changes from LPAR to LPAR and will be altered each time that a machine undergoes a clear/reset operation (which probably means each IPL), no wrapped key should be stored for long-term use. A wrapped key generated on one member of a sysplex cannot be used on any other member of the sysplex, nor on any other LPAR.

3.7 Exploiters of CPACF protected keys

The following products are expected to make use of protected keys:

- DB2/IMS™ Encryption Tool
- Java
- z/OS Encryption Facility

The following products currently make use of CPACF clear key processing:

- DB2/IMS encryption tool
- DB2® built in encryption
- Comm Server: IPsec/IKE
- System SSL
- Kerberos
- DFDSS Volume encryption
- Java
- Encryption Facility

3.8 Summary

In summary:

- CPACF has supplied clear key encryption capabilities for many years, starting with the z990 processor.
- CPACF supplies synchronous cryptographic operations via machine instructions.
- CPACF does not supply any asymmetric encryption capability.
- Starting with the z10 with GA3 microcode, CPACF can provide a high-performance encryption capability with protected keys, that prevents z/OS programs from accessing clear key values.

4

z10 Cryptographic configuration

This chapter introduces the highlights of the z10 Cryptographic configuration and points out the significant changes since the System z9. In the past we have found that many Cryptographic-related changes required outage for the target LPAR, but some of the settings have been enhanced, and now they can be done dynamically, thus enabling you to run other tasks on the target LPAR.

Most of the Cryptographic configuration-related subjects, such as the basic LPAR Cryptographic definition and TKE Command Enablement and Crypto Type modification, have not been changed functionally, and so we suggest that you see the previous IBM Redbooks publication, *z9-109 Crypto and TKE V5 Update*, SG24-7123.

The operations related to Cryptographic settings that must be done via the Hardware Management Console (HMC) or Support Element (SE) are carried using SYSPROG or SERVICE authorizations and therefore require proper logon.

4.1 Usage domain zeroize

The z10 mainframe provides a new function to zeroize individual Cryptographic Coprocessor domain content. This process clears all master key registers and optionally operational keys stored in the target domain. On the previous version of the mainframe, the zeroize capability was only possible at the Cryptographic card level, and this is still a valid option, as shown later in this section.

To be able to perform this task, you should be in the Single Object of Operation mode on the target mainframe and select from the task list on the right the CPC configuration (Figure 4-1). Double-click the Cryptographic Configuration icon.

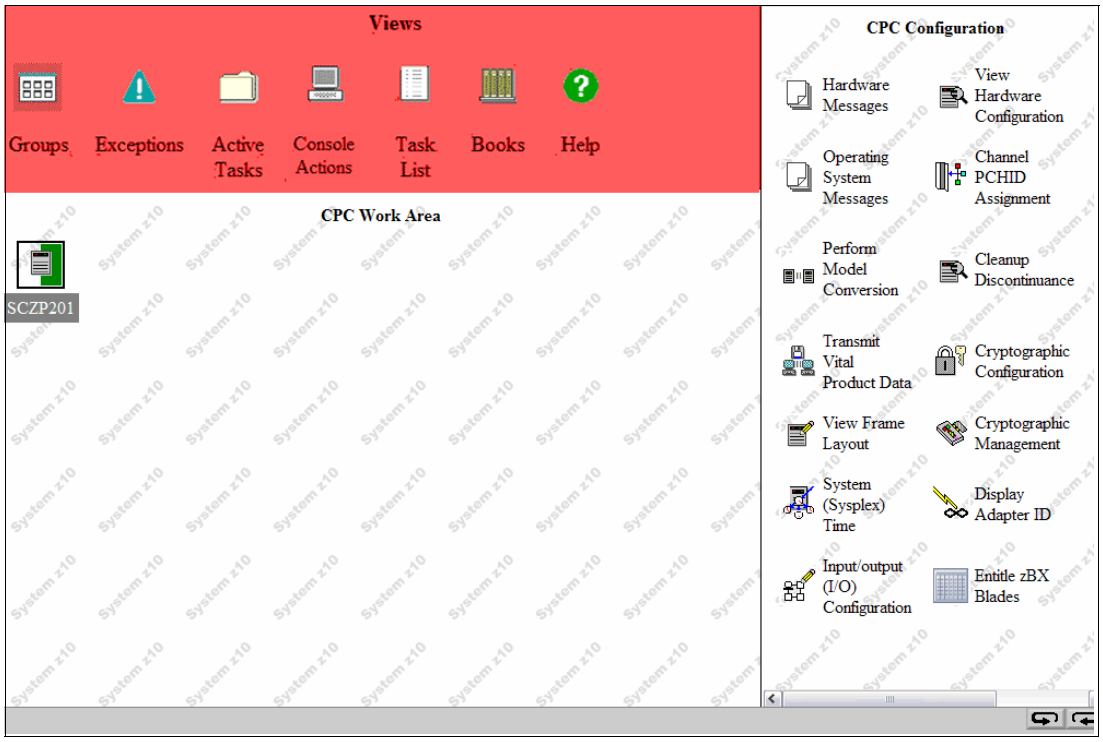


Figure 4-1 Select Cryptographic Configuration on CPC configuration

From the Cryptographic Configuration view select the target Cryptographic Coprocessor that contains the usage domain that you want to zeroize. In our case we selected card 06 (Figure 4-2). Click **Usage Domain Zeroize**.

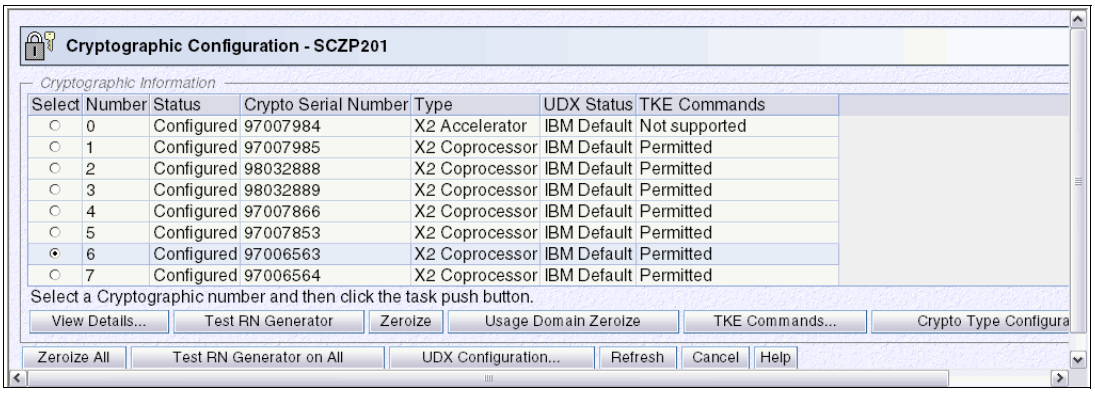


Figure 4-2 The target coprocessor selected for the usage domain zeroize

Note: The usage domain can be zeroized one Cryptographic coprocessor at the time.

The next window (Figure 4-3) displays, and there we select domain number 8 to be zeroized. This view allows the selection of multiple usage domains to be zeroized.

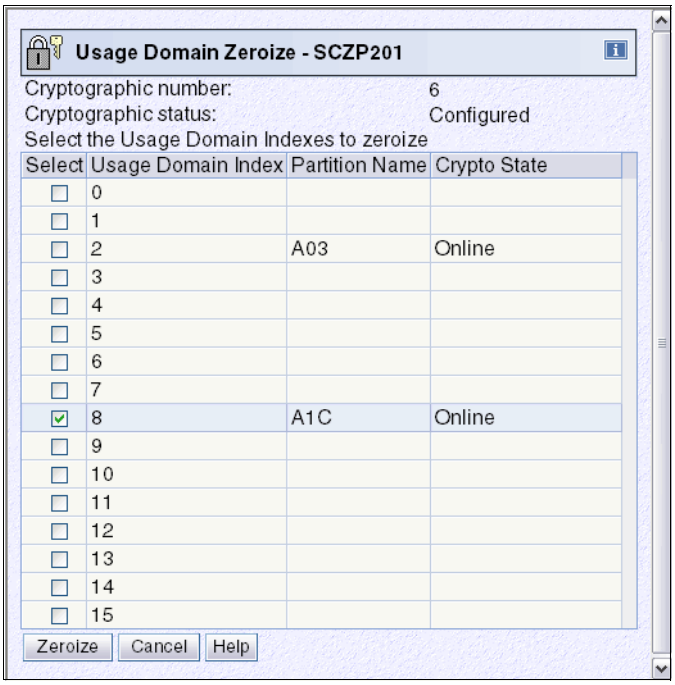


Figure 4-3 Select the domain to zeroize

In our case domain 8 is allocated to LPAR A1C, which causes a warning to appear (Figure 4-4). You need to have the target LPAR connected to the usage domain. You will also need to deactivate the Cryptographic Coprocessor on the target LPAR by using the ICSF panels. This ensures that there is no cryptographic process running on the card when the usage domain is zeroized.

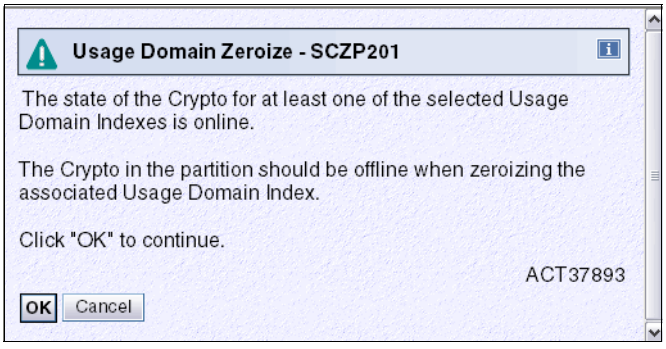


Figure 4-4 Zeroize confirmation

After clicking **OK**, a new window displays to warn you that the zeroize will happen when Zeroize is selected (Figure 4-5).

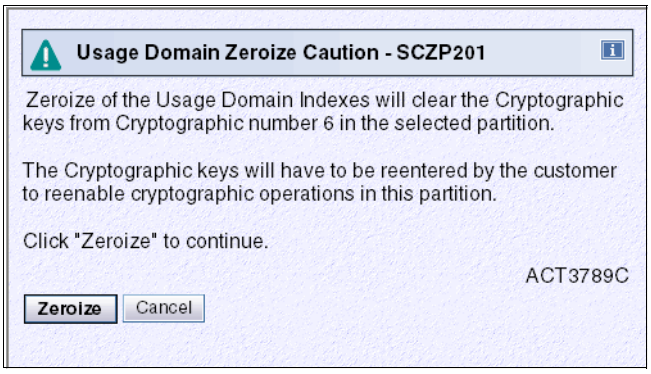


Figure 4-5 The cryptogphic keys swill be cleared

4.2 Viewing and changing LPAR Cryptographic Controls

The z10 mainframe supports a new way to view and change LPAR Cryptographic Controls. This view now gives the summary panel where you can find all LPARs defined on the target mainframe and how many CEX2 and CEX3 processors have been installed. The LPAR Cryptographic Control change allows the making of definition changes without deactivating the target LPAR.

To be able to perform this task, you should be in the Single Object of Operation mode on the target mainframe. Select **CPC Operational Customization** from the task list on the right (Figure 4-6).

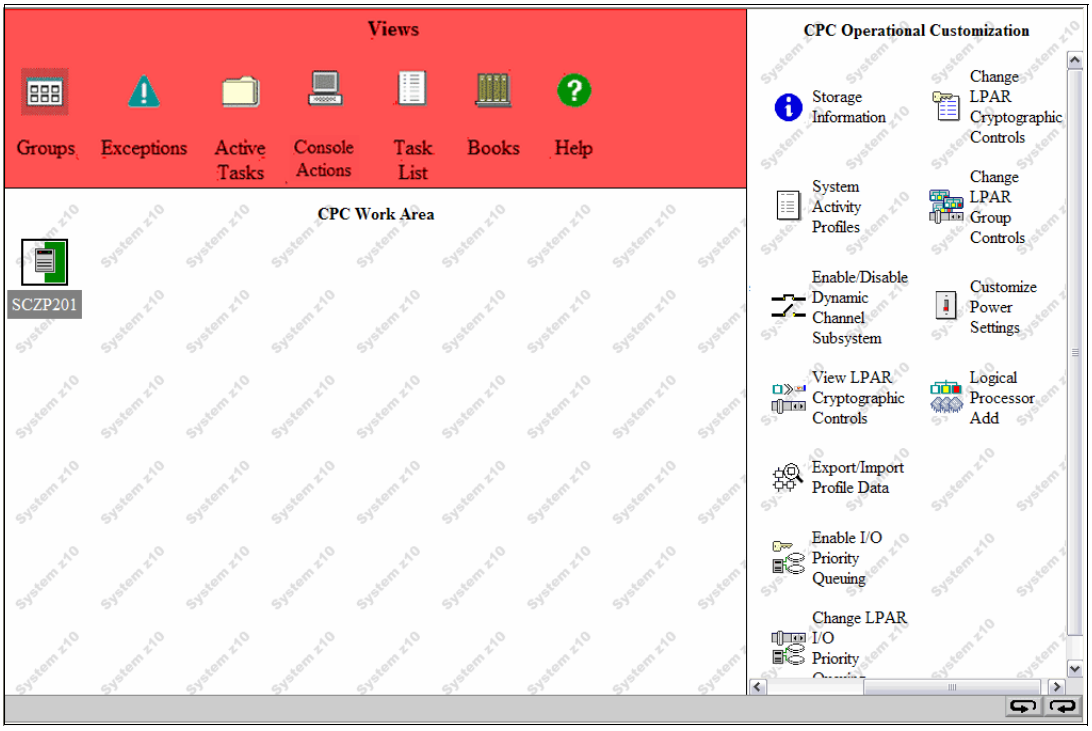


Figure 4-6 Selections on the CPC Operational Customization view

4.2.1 View LPAR Cryptographic Controls

To view the LPAR Cryptographic Controls, double-click the icon on the CPC Operational Customization view. A new window displays (Figure 4-7). You will find all active and inactive LPARs on the summary panel. Those settings related to the inactive LPARs are the same as those defined on the image activation profile.

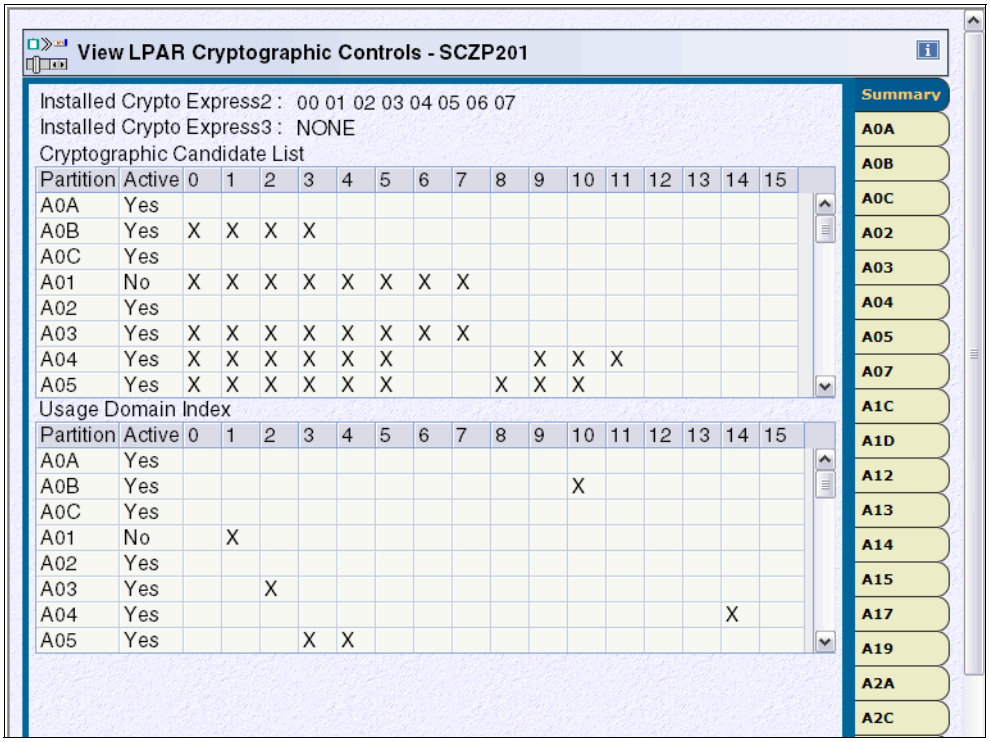


Figure 4-7 LPAR Cryptographic Controls view

When selecting from the tabs on the right of the summary view, an individual LPAR view will be shown (Figure 4-8).

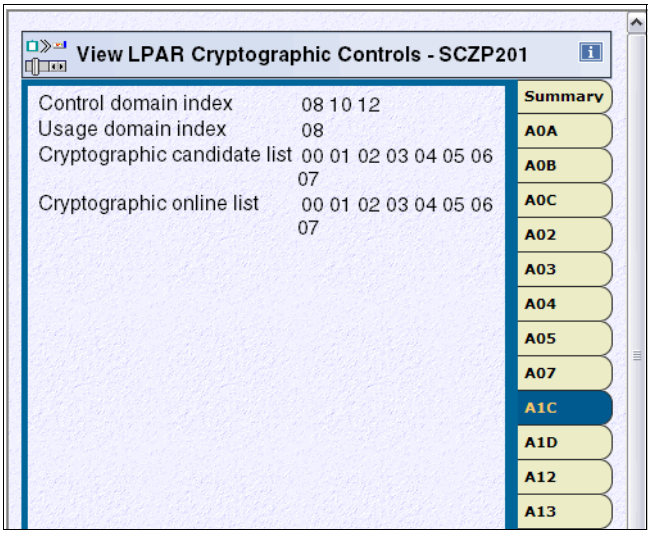


Figure 4-8 Cryptographic Controls for selected LPAR

4.2.2 Dynamically changing LPAR Cryptographic Controls

To modify LPAR Cryptographic Controls, select the target LPAR on the CPC Work Area view and double-click the Change LPAR Cryptographic Controls icon on the CPC Operational Customization view. A new LPAR specific window opens (Figure 4-9).

From this view it is possible to change Cryptographic controls. Clicking Change Running System or Save and Change causes the running system settings to be changed dynamically. Selecting Save to Profile effectively is the same as making changes to the LPAR image profile and then deactivating, activating, and IPLing the target LPAR.

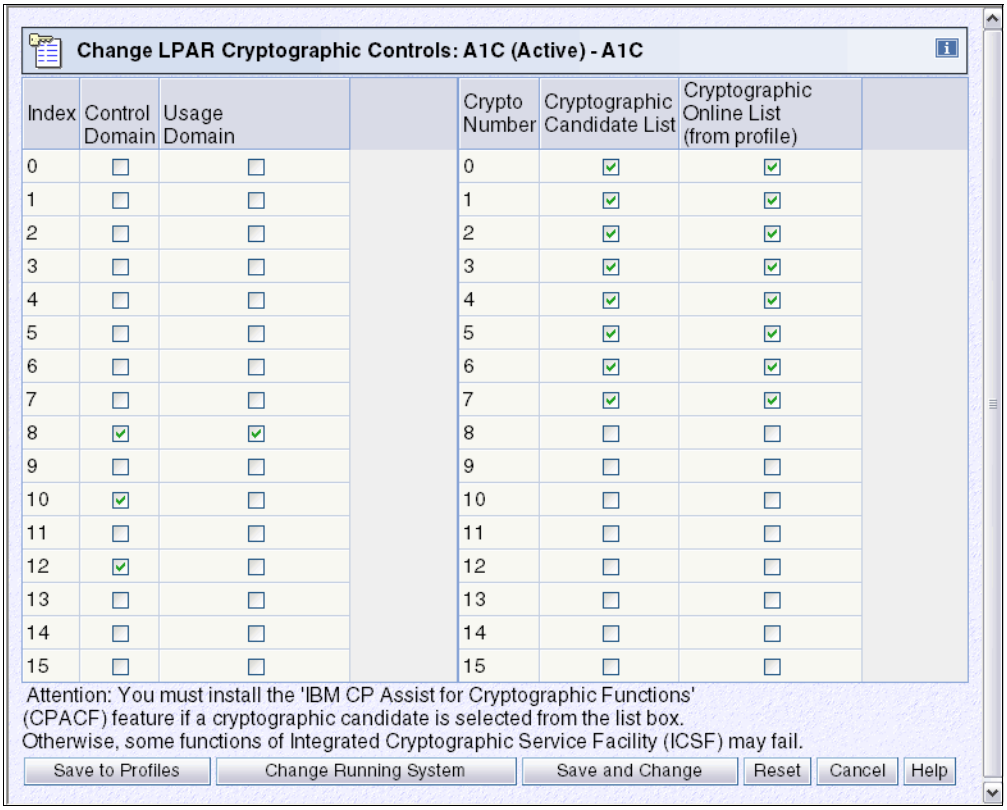


Figure 4-9 Change LPAR Cryptographic Controls view

Special caution should be given to the following change possibilities:

- If changing control domain settings, always have the same control domain index selected as on the usage domain index value.
- If changing the usage domain index value, do not have ICSF running on the running LPAR. You are not allowed to use the same usage domain index value on two running systems.
- If changing the Cryptographic Candidate list content, have at least those Cryptographic cards selected that are defined on the Cryptographic Online list.
- If you plan to remove a Cryptographic card from the online list, first deactivate the target card from the ICSF panel and then configure the card to be in offline state.

4.3 LPAR Cryptographic and security definitions

The z10 image profile definitions are done on the z10 in the same way as on the earlier mainframe systems, although the layout has been slightly modified.

In this section we briefly review the Cryptographic definitions view and new definitions on the security selection. To begin, select the target LPAR and double-click the **Customize/Delete Activation Profile** task on the CPC Operational Customization task list (Figure 4-10).

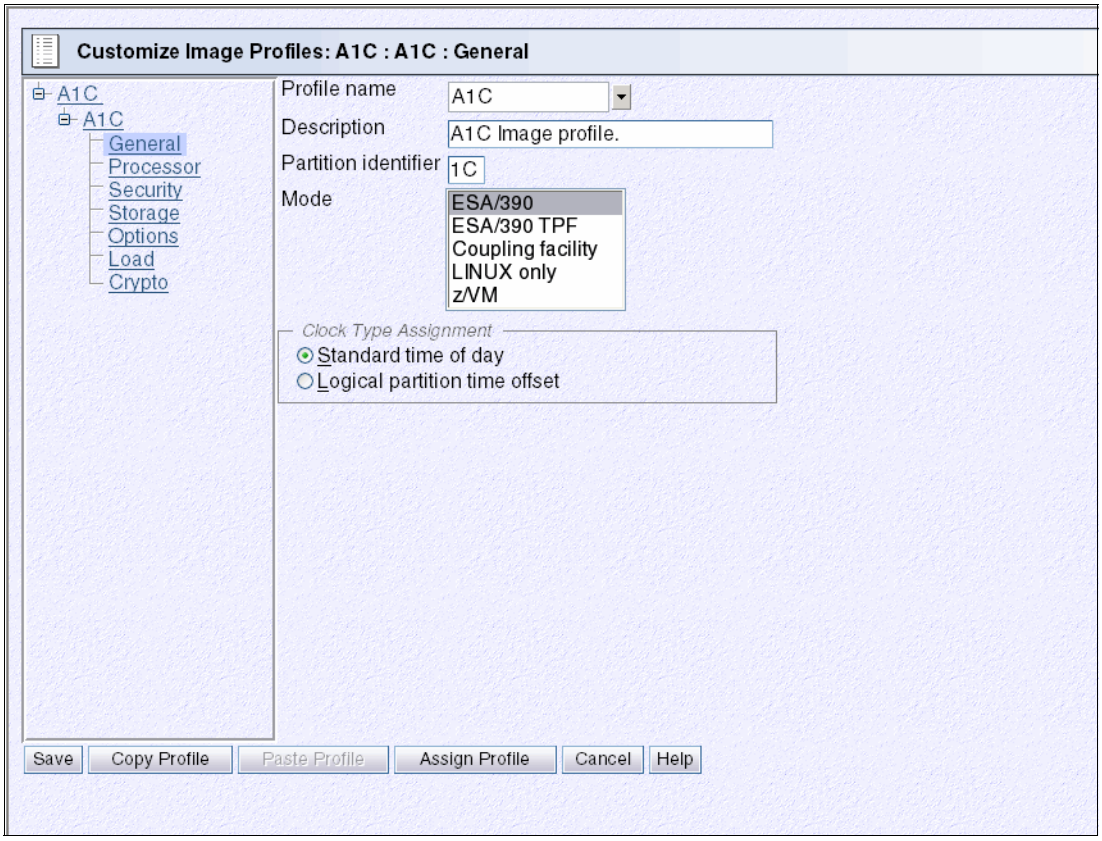


Figure 4-10 Image Profile Customization view

4.3.1 LPAR Cryptographic definitions

To view or modify LPAR Cryptographic settings, select the **Crypto** tab in the Image Profile Customization view (Figure 4-10).

The next panel (Figure 4-11) displays. Here you can define which usage domain is used to store master key values. The control domain index indicates which usage domain can be controlled if this LPAR is used as TKE host LPAR.

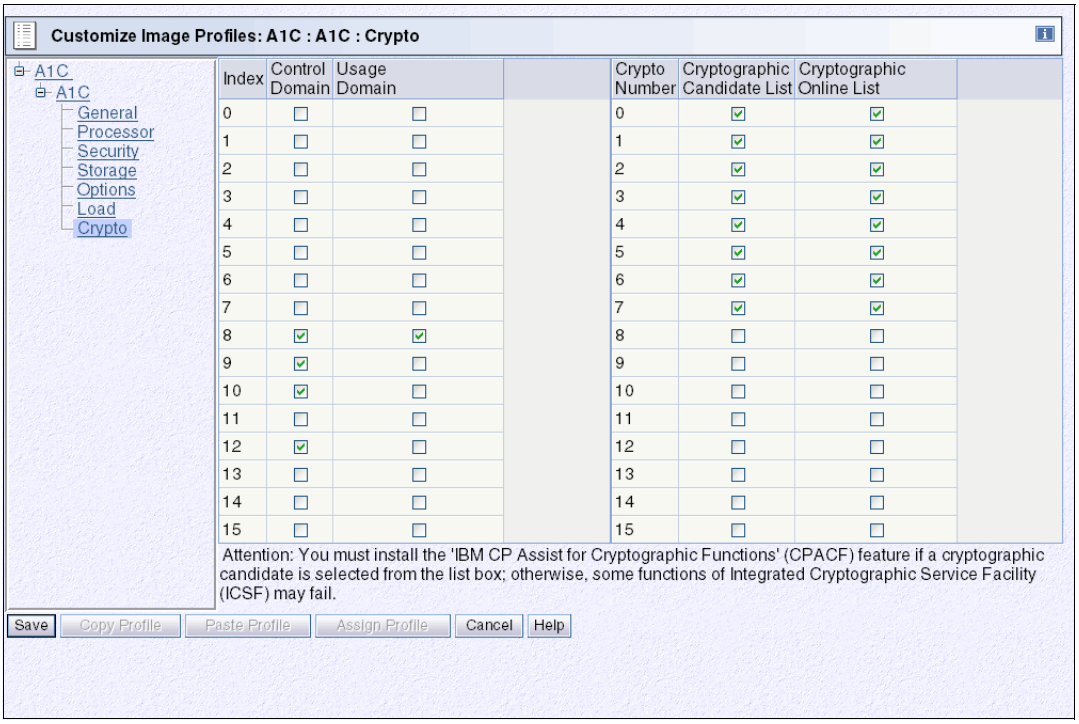


Figure 4-11 Activation profile Cryptographic definitions

On the Cryptographic Online list, you will define those Cryptographic cards that are brought online when the LPAR is activated and IPLed. On the Cryptographic Candidate list you will define the same Cryptographic cards that are defined on the online list. All other changes can be done using the z10 dynamic capability, as shown earlier in this chapter. All the settings defined above should be saved, and then when this LPAR is later deactivated, reactivated, and then IPLed, the changes will become active.

4.3.2 LPAR security settings

The LPAR security settings changes are related to the protected key functionality. As described in Chapter 3, “CP Assist for Cryptographic Functions (CPACF)” on page 51, these keys are wrapped using a wrapping key that is generated when a clear/reset is performed.

For the wrapping key generation purposes, you must select which type of the wrapping key is needed. This setting can be found on the under the security option of the Image Profile Customization page. Click **Security** and a new window (Figure 4-12) displays. To select which wrapping key modes are to be available, you can mark AES, DEA, or both.

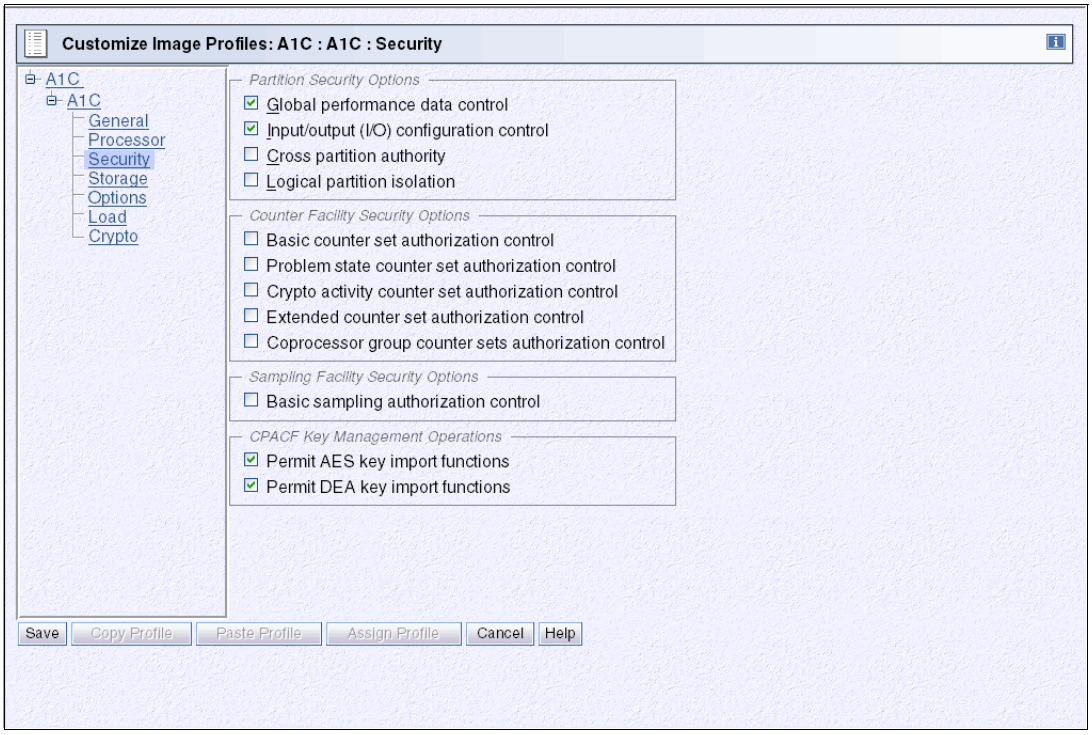


Figure 4-12 The base for the protected key usage on the LPAR

We suggest marking both options because wrapping key generation requires LPAR to be IPLed. (A clear/reset requires an IPL.) Control the usage of this capability with RACF profiles.

Note: This capability is only available on the z10 GA3.

In the same way as other Cryptographic Configuration selections, the definition of the wrapping key options can be done dynamically by selecting the **Change Logical Partition Security** task, which is available from the CPC Operational Customization tasks list to display and modify security controls for an LPAR.

Figure 4-13 shows our test mainframe settings.

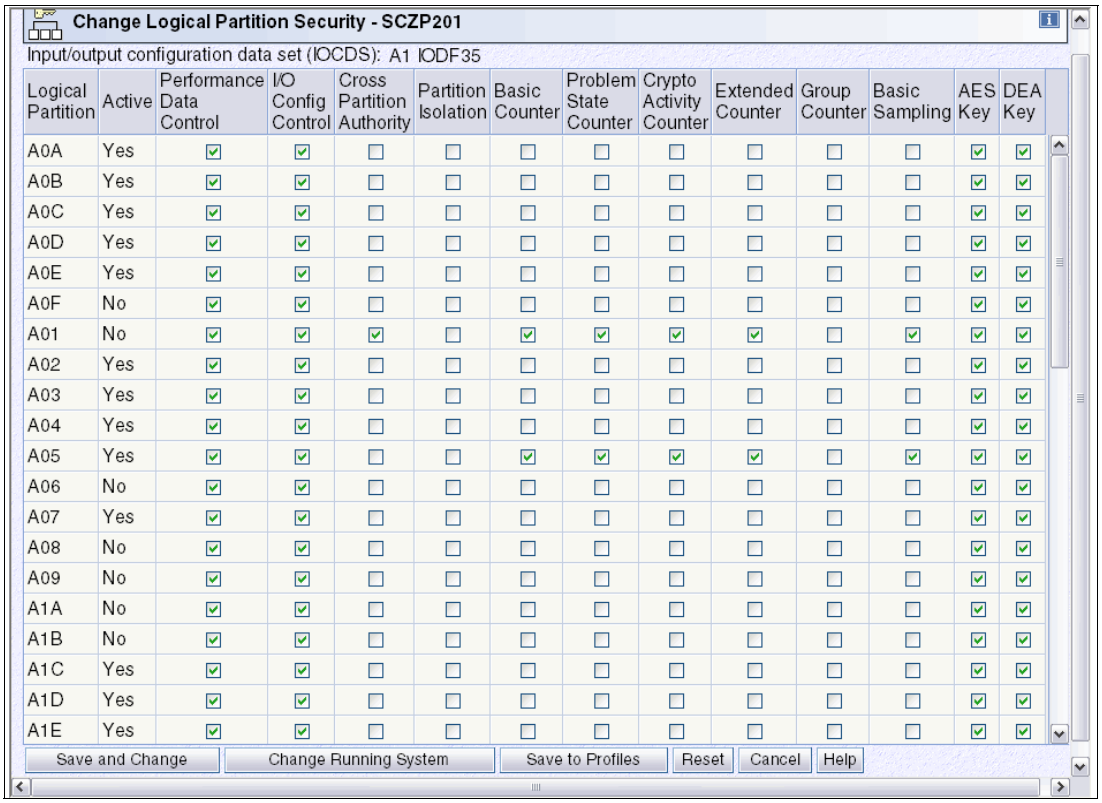


Figure 4-13 Change or view logical partition security

5

Integrated Cryptographic Service Facility (ICSF): Recent changes

This chapter contains the following:

- A table of the various levels of ICSF that are available, and the functions introduced at each level
- A description of the modes of operation of ICSF depending on the hardware available
- A description of the ICSF key store policies and how we tested these functions
- A description of the other new controls introduced at the same time as the ICSF key store policies including the use of the ICSF RACF segment
- Several other smaller topics relating to ICSF and its use

5.1 Introduction

This chapter attempts to shows all the major changes that have been made to ICSF since the HCR7730 version, which was introduced in 2005.

5.2 ICSF levels and capabilities

Table 5-1 is a summary of the capabilities of the levels of ICSF that have been made available since HCR7730. Further details on levels and support dates can be found at:

http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/TD103782

Table 5-1 ICSF levels since HCR7730

ICSF level	Delivered with	Highlights	Server hardware supported	Crypto hardware supported	Minimum z/OS level supported
HCR7730	Web deliverable #5	System z9 support, Sysplex support, System z9 support	z800, z900, z890, z990, System z9, System z10	CCF, PCICC, PCICA, PCIXCC, Crypto Express 2	z/OS 1.6
HCR7731	z/OS 1.8, web deliverable #6	PCI-X Adapter Coprocessor and Accelerator, CPACF enhancements, Remote Key Loading, ISO 16609 CBC Mode TDES MAC	z800, z900, z890, z990, System z9, System z10	CCF, PCICC, PCICA, PCIXCC, Crypto Express 2	z/OS 1.7
HCR7740	z/OS 1.9	z10 BC support, PKCS #11 services CSFPTRC CSFPTRD CSFPTRL CSFPSAV CSFPGAV, cipher feedback mode support in CSNBSYE and CSNBSYD	z800, z900, z890, z990, System z9, System z10	CCF, PCICC, PCICA, PCIXCC, Crypto Express 2	z/OS 1.9
HCR7750	z/OS 1.10, web deliverable #7	Cryptographic exploitation of z10 BC, ISO Format 3 PIN blocks, 4096-bit RSA keys, CPACF support for AES-192 AES-256 SHA-384 and SHA-512, reduced support for retained key in ICSF, random number generate long, CEX2 dynamic add	z800, z900, z890, z990, System z9, System z10	CCF, PCICC, PCICA, PCIXCC, Crypto Express 2	z/OS 1.7
HCR7751	z/OS 1.11, web deliverable #8	Secure key AES including new AES master key, 13 - 19 digit PAN data support, Crypto Query service, Key store policy, IPv6 support (AES MAC), new SMF Type 82 subtypes,	z800, z900, z890, z990, System z9, System z10	CCF, PCICC, PCICA, PCIXCC, Crypto Express 2	z/OS 1.7

ICSF level	Delivered with	Highlights	Server hardware supported	Crypto hardware supported	Minimum z/OS level supported
HCR7770	(z/OS 1.12) web deliverable #9	Protected key CPACF, Crypto Express3 and Crypto Express3-1P, Enhanced PKCS #11 support including Elliptic Curve Cryptography (ECC) support, updates to ICSF query calls	z800, z900, z890, z990, System z9, System z10	CCF, PCICC, PCICA, PCIXCC, Crypto Express 2, Crypto Express3	z/OS 1.9

5.3 Installation issues

Starting with ICSF level HCR7770, there is a new started procedure for ICSF. The first program to be executed on the JCL changes from CSFMMAIN to CSFINIT. In addition, this new program, CSFINIT, is made non-swappable and noncancelable by default. An attempt to execute the old CSFMMAIN produces a message CSFM022E saying that CSFINIT should be used:

```
CSFM022E ICSF TERMINATING. THE USE OF CSFINIT REQUIRED IN THE STARTED
TASK PROCEDURE.
```

ICSF has been improved in terms of consistency of delivery of messages. The CSFLIST DD statement is no longer required. The output previously placed in CSFLIST is now displayed on the system log.

Also note that the REGION statement has been set to REGION=0M. This is now a recommended value for all current levels of ICSF.

The installation of ICSF that we used for testing also had the following maintenance applied:

```
UA51563 UA51708 UA51711 UA51800 UA51886 UA52279 UA52385 UA52457 UA52918 UA52972
UA53085 UA53250 UA53376 UA53484 UA53595
```

When installing any new level the install bucket for ICSF should be examined.

5.4 Modes of operation

There are three theoretical ways in which ICSF can be configured. We go through these in this section.

5.4.1 ICSF with CEXnC

This is the normal method, where ICSF has access to one or more Crypto Express devices (CEX2C or CEX3C) and is able to process the full range of requests by making use of:

- Software internal processes
- CPACF-enabled processes
- CEXnC processes

Certain processes are handled via software routines. Examples of this are the MD5 and RIPEMD-160 hashing methods supplied by the API CSNBOWH.

The second group of operations are those handled by CPACF instructions. Typical of these are routines such as CSNBSAE and CSNBSYE for performing clear-key encryption using TDES or AES encryption algorithms.

The third group of services comprises those that require being passed to the Crypto Express device.

Initialization options

With the introduction of HCR7751 came the introduction of the AES master key, which is used to secure AES keys in the CKDS. Depending on what master keys are specified, it is possible to initialize the CKDS on three different modes:

- DES-MK only
- AES-MK only
- DES-MK and AES-MK

To make use of DES secure keys, or AES secure keys, the corresponding master key must be set. If the master key is not set, then only clear keys of that type can be stored.

5.4.2 ICSF with CPACF

Starting with HCR7751, it is possible to initialize a CKDS data set and run ICSF without having any Cryptographic processor available. (This support is available on all System z processors except the z900 and the z800. These require the CCF to be available to initialize the CKDS.)

This level of support is useful only if there is a desire to use clear keys and have them stored in the CKDS. This is useful if clear key support is adequate for your security and privacy needs, and you want to make use of encryption. It is also possible to use protected keys in such a situation.

One software product that could make use of this is the product is IBM Data Encryption for IMS and DB2 Databases Version 1.1 - 5655-P03. This product can be used to encrypt data using clear keys stored in the CKDS. However, in releases of ICSF prior to HCR7751, the CKDS has to rely on the presence of a cryptographic processor to perform the initialization, and subsequently the cryptographic processor is not needed, as the clear keys will be passed to CPACF clear key instructions to perform the encryption and decryption.

A CKDS initialized in this way will differ from a more normal CKDS in that there is no information about master key verification patterns within the CKDS header records.

If a CKDS is used in this way, it is vital that the CKDS be protected such that it cannot easily be read. RACF controls should be used to prevent READ access to the CKDS VSAM data set.

If a CKDS is initialized in this way, then there is no formal method to convert this to the more standard CKDS if a cryptographic feature is purchased and used on the LPAR. However, there is a method of performing such a conversion. The tools needed are supplied in a package called `CKDSMERG.zip` and are provided at the following website:

`http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/PRS1953`

Chapter 8 in the *ICSF System Programmer's Guide*, SA22-7520-14, documents the process for initializing the CKDS without a Cryptographic Coprocessor being present. This is a simple process and involves creating a CKDS and PKDS in the normal way, and then using option 2.1 within the ICSF panels to specify the name of the CKDS and initialize it.

We performed this initialization on a system running HCR7751 and having no Cryptographic Coprocessors. Figure 5-1 shows the startup messages for ICSF.

On an LPAR that has Crypto Express devices, we deactivated them, and then tried to initialize a *clear-key* CKDS, but this is not allowed. ICSF appears to detect the presence of the Crypto Processors and so assumes that the CKDS should undergo a full initialization.

```
CSFM607I A CKDS KEY STORE POLICY IS NOT DEFINED.
CSFM607I A PKDS KEY STORE POLICY IS NOT DEFINED.
CSFM610I GRANULAR KEYLABEL ACCESS CONTROL IS ENABLED.
CSFM611I XCSFKEY EXPORT CONTROL FOR AES IS DISABLED.
CSFM611I XCSFKEY EXPORT CONTROL FOR DES IS DISABLED.
CSFM612I PKA KEY EXTENSIONS CONTROL IS DISABLED.
CSFM101E PKA KEY DATA SET, WTSCPLX1.SC50.SCSFPKDS IS NOT INITIALIZED.
CSFM506I CRYPTOGRAPHY - THERE IS NO ACCESS TO ANY CRYPTOGRAPHIC
COPROCESSORS OR ACCELERATORS.
CSFM012I NO ACCESS CONTROL AVAILABLE FOR CRYPTOZ RESOURCES. ICSF
PKCS11 SERVICES DISABLED.
CSFM122I PKA SERVICES WERE NOT ENABLED DURING ICSF INITIALIZATION.
CSFM001I ICSF INITIALIZATION COMPLETE
CSFM126I CRYPTOGRAPHY - FULL CPU-BASED SERVICES ARE AVAILABLE.
```

Figure 5-1 ICSF log messages

5.4.3 ICSF without CPACF

It is theoretically possible to run ICSF without either CPACF or a Crypto Express device. In this case, the only services available are those that are entirely software driven, such as hashing using MD5 or RIPEMD-160, as mentioned above.

As we have no access to a System z processor that does *not* have a CPACF feature enabled, we have no way of testing this.

Even though this is a theoretical option, we do not recommend it. CPACF is a no-charge feature that is available in most geographies.

5.5 Key store policies

This section examines the recent introduction of key store policies and attempts to show why they were introduced, what can be achieved by using them, and how to use them.

Note: If you are planning to make use of these policies and want to read the IBM ICSF manuals, then we suggest that you use the manuals for HCR7770 rather than those for HCR7751. The HCR7770 manuals have an excellent explanation and provide more details.

Some of the controls detailed below are not truly key store policies, but they are very closely aligned. The ICSF manual *z/OS Cryptographic Services Integrated Cryptographic Service Facility - Administrator's Guide*, SA22-7521-14, very carefully describes which of these controls is a formal Key Store policy. Refer to Chapter 4 of that manual for more details.

5.5.1 The need for a a key store policy

The ICSF level HCR7751 introduced several sets of new RACF controls on the use of encryption keys. These were introduced to address very specific challenges over the use of encryption keys within large mainframe computing installations.

Let us first look at those challenges and then see how each one has been addressed. But first we provide a little background.

Background information about the CKDS and key labels

The CKDS is the key store for symmetric keys. Each key in the store is held in a token, which is 64 bytes in length. Also associated with the token is a label. The label is also 64 bytes in length. The key store data set (a VSAM data set) is keyed (in the VSAM sense) using this label so that a reference to a label is easily turned into a reference to a key token.

Note: The CKDS VSAM data set is actually keyed (in the VSAM sense) on both the key label and the key type. The key type is an 8-bytes field, so the total (VSAM) key length is 72 bytes.

When ICSF initializes, it loads the entire key store into a data space for high-speed access. From this point any reference to a key label can easily be translated into a reference to a key token.

For security purposes, each time that a key label is used in an API, a check is made in the RACF class CSFKEYS[1]. Let us assume that I made reference to a key such as LENNIE.SECURE.AESKEY01. This key label could be used in an API such as CSNBSAD to decrypt data. The REXX statement one could use to call CSNBSAD might look something like this[2]:

```
address linkpgm 'CSNBSAD',
                'ex_rc',
                'ex_rs',
                'exit_data_length',
                'exit_data',
                'rule_array_count',
                'rule_array',
                'key_length',
                'key_identifier',
                'key_parms_len',
                'key_parms',
                'block_size',
                'init_vector_len',
                   'init_vector',
                   'chain_data_len',
                   'chain_data',
```

[1] While we have discussed a RACF check, this is really a SAF check, and other security products can also be used to respond to the SAF check.

[2] This statement is taken from the REXX program REXPK001, which can be found in Appendix A, "Sample programs in assembler and REXX" on page 261.

```
'cipher_text_len',
'cipher_text',
'clear_text_len',
'clear_text',
'opt_data_len',
'opt_data'
```

The key_identifier field is in bold. This field contains the 64-byte label of the key we are about to use.

When this API is used a check is made using a RACROUTE REQUEST=FASTAUTH macro. This security check is passed to RACF and request the following:

- Resource class = CSFKEYS
- Resource name = LENNIE.SECURE.AESKEY01
- Access level = READ

This request is made using the FASTAUTH type request for speed purposes.

Note: Another check is also made during the processing of the ICSF request. This is for:

- Resource class = CSFSERV
- Resource name = CSFSAE
- Access level = READ

However, this is not relevant to the discussion of keys store policies.

Only if the necessary access to the key has been granted by RACF will the request proceed. If the access has not been granted, then the request will be refused with return code 8 and reason code X'3E84'. In addition, an audit record might be produced.

Using key tokens instead of labels

We have seen that a key label can easily be translated to a key token. However, until recently (ICSF level HCR7751), the converse was not true. If within ICSF a reference is made to a token, there was never any need to relate that reference to a label. So why would we need to be able to do the reverse lookup? Because the token that corresponds to the key above has the following contents (on our test system at least):

```
X'01000000040OC0247BF8BD0F2630EE49'
X'983C1E2BC922AF8A6D9A1A21AC35659E'
X'1088D5EFBE392C3681C7A3A4CFC10BE7'
X'00000000000000000100002043A36AC0'
```

As can be seen, this is a 64-byte string, but it has little structure visible to the eye. The important point here, however, is that if we specify this in the key_identifier field in the previous example, then the access check to RACF is *not* made.

Challenge number 1 is to ensure that when we use a key token rather than a key label, the same RACF security controls are in place.

Default token checking

In the example above we showed a key token that was derived from a record in the CKDS. It had an exact correspondence to a key label.

But what happens if we simply generate a key token (using the CSNBKGN API, for example) and then try to use it? It is highly unlikely that this will match an existing key token, so there will be so security on the use of that token.

Our second challenge is therefore to ensure that if a token is used that is *not* associated with an existing key label, and that some security is still applied to its use.

Duplicate token prevention

Now let us see what might happen if a tricky programmer came along and considered the following.

The programmer wants to use the key LENNIES.SECURE.AESKEY01, but he does not have access. He is able to obtain a copy of the key token instead, and tries to use that. However, he is prevented from doing that too (we will see how later).

He then proposes to take the token and write a NEW CKDS key label, with an entirely different name (called, for example, PATRICKS.SECURE.AESKEY01) and then he will be able to use the token. Once he has a copy of the token, he can use the ICSF APIs CSNBKRC and CSNBKRW to create such a label[3].

The nub of this type of processing is that a key can be created with the same value under two different labels.

Our next challenge is to prevent a single key token from being used via two distinct key labels.

Implementing access levels for key checking

Another challenge facing key management staff is that, as we saw above, to use a key, a user needs access to it under the RACF class CSFKEYS.

Let us contrast this with z/OS data set access controls.

If a user has access to a data set then he can have access at various access levels. Table 5-2 shows the access levels available.

Table 5-2 Access levels

Access level	Meaning for data set	Notes
NONE	No access is ever granted	
EXECUTE	Can execute programs	This type of access applies only to a specific class of data sets that contain programs (load modules or program objects) to be executed.
READ	Can read the contents	
UPDATE	Can change the contents	Using this level of access it is possible to entirely destroy the data within the data set.
CONTROL	Can use CI access	This is a special access level associated with VSAM data sets.
ALTER	Can create, rename, delete.	This level of access is need manipulate the entire data set including contents. Note**:** For discrete profile, also allows altering of the access list and deletion of the profile entirely.
OWNER	Full control	This is *not* really an access level. However, having ownership of the profile that controls a data set allows access to manipulate the security profile that controls access.

[3] CSNBKRW is passed a token, but this token is NOT checked to see if it matches an existing label.

However, you will have seen that the access check for the use of key labels (which is made in the class CSFKEYS), *only ever uses the access level READ*.

While it may seem odd to you, the access level required to READ a key token and the access level required to *write* (or re-write) a key token are exactly the same. They both use *read*. So if I grant you access to use a key label (so that you can decrypt some data), I am also granting you access to change that key token, which means that you are preventing others from decrypting the data, even though I have granted them that access too.

Challenge number 4 is to prevent users needing to decrypt or encrypt data from being able to alter keys.

When we get to describing that checking mechanism we will produce a table similar to the one above, but for ICSF key labels.

Symmetric key export controls

A further shortcoming with regards to accessing keys is that sometimes it becomes necessary to allow a key to be used (that is, READ access is granted) so that it can be exported under a transport key. This type of operation can be achieved using the CSNBKEX API. However, it is not possible to separate the access to the key from the ability to export it.

Challenge number 5 is to enable enhanced control of the ability to export keys.

PKA key management extensions control: Part 1

The next challenge relates to the use of asymmetric keys and how they are used.

Asymmetric keys are frequently associated with PKI certificates, and by using the appropriate parameters on RACDCERT ADD commands (ICSF and PCICC) it is possible to store the asymmetric keys associated with a certificate within the PKDS.

Each certificate has settings within it that show how it can be used. For example, it might be used for signing other certificates, or it might be used in setting up SSL communications. If the key is always kept with the certificate then this use is enforced.

However, if the keys are stored in the PKDS (a far more secure place to store them), then the keys can be potentially used for something other than the original intention.

Hence, challenge number 6 is to enable controls on the use of asymmetric keys, in terms of the operations that it performs.

This function is only available with ICSF level HCR7770 and with z/OS V1R10 or later.

PKA key management extensions control: Part 2

The second part of the PKA key management extensions relates to which asymmetric keys are allowed to manipulate which symmetric keys.

Thus, we will have a control that can prevent an asymmetric key from being used to export a specific symmetric key. This is the most specific of these new controls on the exporting of symmetric keys by asymmetric keys.

This is challenge number 7, the final challenge.

5.5.2 Key store policy checking

Key store policy options are enabled by creating RACF profiles in the XFACILIT class. The access list of these profiles is not used. It is merely the existence of the profile that alters the behavior of ICSF.

In some cases there are two profiles for a given facility. In this case, the first profile ends with the characters `FAIL`, while the second profile ends with the characters `WARN`. This is designed to be used so that the policy can be placed into warning status prior to full enablement.

The descriptions given above talk about the CKDS. However, these challenges apply to the PKDS as well, so there are controls for that key store as well.

When the RACF profiles are created, ICSF is notified dynamically and then activates the policies immediately.

5.5.3 Token checking

Our first challenge is that for many of the ICSF APIs it is possible to specify either a key token or a key label, and we want to ensure that if a key token matches the key token associated with an existing key label, then the key label should be used for checking access to the key token.

The checking that takes place here makes use of a hash value that ICSF calculates for each key in the key store. This mechanism provides a fast way of determining which key label (if any) is associated with the key token.

Enablement

To enable this checking, one or more of the following profiles must be created in the XFACILIT class:

```
CSF.CKDS.TOKEN.CHECK.LABEL.FAIL
CSF.CKDS.TOKEN.CHECK.LABEL.WARN
CSF.PKDS.TOKEN.CHECK.LABEL.FAIL
CSF.PKDS.TOKEN.CHECK.LABEL.WARN
```

These can be created using a RACF command such as:

```
RDEFINE XFACILIT CSF.CKDS.TOKEN.CHECK.LABEL.WARN UACC(N)
```

For either of the keys stores, if both the FAIL profile and the WARN profile exist, then checking takes place in FAIL mode.

Simply by issuing the command above, we get the following message issued by ICSF:

```
CSFM608I A CKDS KEY STORE POLICY IS DEFINED.
```

If I delete the above profile I get the message:

```
CSFM607I A CKDS KEY STORE POLICY IS NOT DEFINED.
```

Checking the process

To check this we created REXX programs to create a key and store it in the CKDS. We also had a routine to read the key and store the key token elsewhere.

Lastly, we denied ourselves access to the key label.

We then tested with no token checking policy in place.

We found that an attempt to use the key label with CSNBENC and CSNBDEC ended with return code 8 and reason code X'3E84'. However, an attempt to perform the same process with the key token succeeded.

We then turned on key label checking at WARN mode by issuing the following:

```
RDEFINE XFACILIT CSF.CKDS.TOKEN.CHECK.LABEL.WARN UACC(N)
SETROPTS RACLIST(XFACILIT) REFRESH
```

Then we tried the test again using the token. We received the following warning messages:

```
ICH408I USER(LENNIE  ) GROUP(SYS1    ) NAME(LENNIE DYMOKE-BRADSH)
  LENNIE.DESKEY.TEST001 CL(CSFKEYS )
  INSUFFICIENT ACCESS AUTHORITY
  ACCESS INTENT(READ   )  ACCESS ALLOWED(NONE   )
```

But the encryption and decryption succeeded.

We then set the checking to FAIL mode by issuing the following:

```
RDELETE XFACILIT CSF.CKDS.TOKEN.CHECK.LABEL.WARN
RDEFINE XFACILIT CSF.CKDS.TOKEN.CHECK.LABEL.WARN UACC(N)
SETROPTS RACLIST(XFACILIT) REFRESH
```

When we executed the test with the key token, we received return code 8 reason code X'0BF7', thus demonstrating that the token checking was working correctly.

We then changed the key associated with the key label LENNIE.DESKEY.TEST001 and tested with the original key token. Predictably, in this test the encryption succeeded.

Extra notes

Note the following:

- Keys of different types, but with the same clear key values are allowed. This is consistent with the need to generate key pairs of various types, such as EXPORTER and IMPORTER or IPINENC and OPINENC keys.
- The example we used specified the use of SETROPTS commands to refresh the XFACILIT class whenever changes are made. However, it is not necessary to have this class RACLISTed, so the SETROPTS commands might not be needed in your installation.
- If the token that is passed to ICSF matches more than one key token in the CKDS, then the least restrictive key label will be used for access checking. We did not test this.
- The checks that we made were all performed using the CKDS and symmetric token. The same rules should apply to the PKDS and asymmetric token, but we did not test this.

5.5.4 Default token checking

In the last test in the previous exercise we performed above, we used a token that was no longer associated with any key in the CKDS. We found that we were not blocked from using this token.

The purpose of the next check is to enable security to be applied to the use of a key token that has no match to any key label in the CKDS (or PKDS).

This check makes use of the same hashing values that we spoke of before. However, in this case the checking will be triggered by a hash value matching *none* of the values in the key store.

Enablement

This check is enabled by the creation of one of the following profiles:

```
CSF.CKDS.TOKEN.CHECK.DEFAULT.LABEL
CSF.PKDS.TOKEN.CHECK.DEFAULT.LABEL
```

Once the above profiles have been created, we also need to create a profile to act as the default key label for the key token. The names to use are specified as:

```
CSF-CKDS-DEFAULT
CSF-PKDS-DEFAULT
```

These resources must be defined in the CSFKEYS class. To define a default profile with no one on the access list for symmetric keys, use the following commands:

```
RDEFINE XFACILIT CSF.CKDS.TOKEN.CHECK.DEFAULT.LABEL UACC(N)
RDEFINE CSFKEYS CSF-CKDS-DEFAULT UACC(N)
SETROPTS RACLIST(XFACILIT,CSFKEYS) REFRESH
```

Checking the process

We defined these profiles and then examined the ICSF log. No messages were produced. We then ran a test that attempted to use a key token with no corresponding key label and found that access to the key token was *not* denied.

However, this is because for this check to be in use, you also have to have the above token checking key store policy active. We enabled the token checking key store policy and repeated our test, and we received the following messages:

```
ICH408I USER(LENNIE  ) GROUP(SYS1    ) NAME(LENNIE DYMOKE-BRADSH)
  CSF-CKDS-DEFAULT CL(CSFKEYS )
  INSUFFICIENT ACCESS AUTHORITY
  ACCESS INTENT(READ   )  ACCESS ALLOWED(NONE   )
```

Access to the key token was denied with return code 8 and reason code X'0BF8'.

Extra notes

Note the following:

- As we found out (and as is documented), the default token processing only works if token checking is active for the key store in question (that is, the CKDS or PKDS).
- However, the token checking does not need to be at FAIL level. Token checking can be at WARN level, and the default token checking will still work. We tested this.
- This control is not actually deemed a key store policy. It is really an addition to the previous key store policy.
- The example we used specified the use of SETROPTS commands to refresh both the XFACILIT class and the CSFKEYS class whenever changes are made. However, it is not necessary to have these classes RACLISTed.

5.5.5 Preventing duplicate tokens

The next control relates to duplicate tokens. This is a situation where two key labels exist, which are associated with the same key token. Note that we are talking about two key labels of the same key type.

Enablement

To enable this key store policy it is necessary to create one of the following profiles in the XFACILIT class:

```
CSF.CKDS.TOKEN.NODUPLICATES
CSF.PKDS.TOKEN.NODUPLICATES
```

The command to activate duplicate token checking is:

```
RDEFINE XFACILIT CSF.CKDS.TOKEN.NODUPLICATES UACC(N)
SETROPTS RACLIST(XFACILIT) REFRESH
```

We deactivated all previous key policies and then issued the above commands. Once this is done the ICSF address space issues the following message:

```
CSFM608I A CKDS KEY STORE POLICY IS DEFINED.
```

We then attempted to make two key labels with the same key token value. The first succeeded, but the second failed with return code 8 and reason code X'0BF9'.

Extra notes

Note the following:

- This control is deemed a key store policy, and hence produces message CSFM608I when activated.
- If duplicate key tokens already exist at the point where this control is activated, then ICSF issues the following message:

  ```
  CSFC0322 DUPLICATE TOKENS FOUND IN DATASET WTSCPLX1.SC49.SCSFCKDS
  ```

- Enabling this control prevents new duplicate keys from being added and identifies whether duplicates already exist. However, to find those duplicates you can run the CSFDUTIL program. The JCL to run this is shown in Figure 5-2, and sample output for the CKDS is shown in Figure 5-3.

```
//FINDDUPS EXEC  PGM=CSFDUTIL
//SYSOUT    DD   SYSOUT=*
//SYSIN     DD   *
 CKDSN(WTSCPLX1.SC49.SCSFCKDS)
/*
```

Figure 5-2 Sample JCL

```
DUPLICATE TOKEN CHECK REPORT                                        DATE: 2010/05/25 (YYYY/MM/DD) TIME: 08:52:19 PAGE:     1

CKDS: WTSCPLX1.SC49.SCSFCKDS

* Group 1: 2 Duplicates
Key Label                                                    Key Type  Cre date  Cre time  Upd date  Upd time
------------------------------------------------------------ --------  --------  --------  --------  --------
LENNIE.DESKEY.TEST001                                        DATA      20100524  18024944  20100524  18024953
LENNIE.DESKEY.TEST002                                        DATA      20100524  18024961  20100524  18024967

*** KDS HAS 8 RECORDS IN TOTAL
*** KDS HAS 5 SECURE TOKENS
*** KDS HAS A TOTAL OF 2 DUPLICATE TOKENS IN 1 GROUP

PROCESSING COMPLETE
```

Figure 5-3 Sample output

5.5.6 Key label access levels

The next challenge is to provide a mechanism to ensure that providing access to a key does enable the changing or destruction of that key.

This set of controls is sometimes called *granular key label access controls*.

To do this we associate different types of operations with a key with the levels of access that are available. (See Table 5-2 on page 90 for details about those access levels as associated with data sets.)

Table 5-3 is he equivalent table of access levels for ICSF key labels and key tokens.

Table 5-3 Access levels

Access level	Meaning for ICSF key label	Notes	New with key label access controls?
NONE	No access is ever granted		NO
EXECUTE	Not used.	Not relevant. Will be treated as NONE.	NO
READ	Can use the key in normal operations.	By example, encryption keys can be used for encryption, MACing keys can be used for MACing, PINGEN keys can be used for PIN generation, and so on.	NO
UPDATE	Can create a label.	This allows the creation of a key label only: CSNBKRC - Key Record Create - CKDS CSNDKRC - Key Record Create - PKDS CSNFKRC - Key Record Create - PKDS	YES
CONTROL	Can WRITE a key to a label and DELETE a label.	This allows the following APIs to be used: CSNBKPI - Key Part Import CSNBKRW - Key Record Write - CKDS CSNDKRW - Key Record Write - PKDS CSNDPKG - PKA Key Generate CSNFPKG - PKA Key Generate CSNDPKI - PKA Key Import CSNFPKI - PKA Key Import CSNDTBC - Trusted Block Create CSNBKRD - Key Record Delete - CKDS CSNDKRD - Key Record Delete - PKDS CSNFKRD - Key Record Delete - PKDS CSNDRKD - Retained Key Delete CSNFRKD - Retained Key Delete	YES
ALTER	Can potentially delete the protection.	This level of access is needed to manipulate the entire data set including contents. Note**:** For discrete profile, ALTER access also allows altering of the access list, the UACC, and deletion of the profile entirely.	NO
OWNER	Full control.	This is *not* really an access level. However, having ownership of the profile that controls access to a key label allows complete access to manipulate the security profile that controls access.	NO

In Table 5-3 on page 96 we included *all* the possible levels of access for key labels with the new controls in place. As you can see, the new levels are at UPDATE and CONTROL. If this new set of controls were not in place, then all those requests types in Table 5-3 on page 96 that are for CONTROL or UPDATE would be granted only READ access.

Note: It might seem more intuitive to be able to update a key with UPDATE rather than CONTROL access, but the reader should remember that READ, UPDATE, and CONTROL are really arbitrary names associated with access levels. They were chosen years ago for data sets, and have been adapted for other resources (in this instance, key labels and key tokens). However, the access levels for NONE, EXECUTE, and ALTER have (or can have) more specific meanings in RACF.

Enablement

To enable this key store policy it is necessary to create one of the following profiles in the XFACILIT class:

```
CSF.CSFKEYS.AUTHORITY.LEVELS.WARN
CSF.CSFKEYS.AUTHORITY.LEVELS.FAIL
```

As you can see, this control is activated for both the CKDS and the PKDS at the same time. However, you have the option to place this control in WARN status before enforcing it.

The command to enable this control in ICSF in WARN mode is:

```
RDEFINE XFACILIT CSF.CSFKEYS.AUTHORITY.LEVELS.WARN UACC(N)
SETROPTS RACLIST(XFACILIT) REFRESH
```

Checking the process

We defined the above profile and then examined the ICSF log. We found that the following message had been issued:

```
CSFM610I GRANULAR KEYLABEL ACCESS CONTROL IS ENABLED.
```

We then granted READ access to our test key labels and attempted to create them with a REXX program that performed a CSNBKGN to create a token, then a CSNBKRC to create the label, and finally a CSNBKRW to write the key token to the key label.

With READ access to the label, the CSNBKRC fails with the following messages:

```
ICH408I USER(LENNIE  ) GROUP(SYS1    ) NAME(LENNIE DYMOKE-BRADSH)
   LENNIE.DESKEY.TEST001 CL(CSFKEYS )
   INSUFFICIENT ACCESS AUTHORITY
   ACCESS INTENT(UPDATE )  ACCESS ALLOWED(READ )
```

The request ended with return code 8 and reason code X'3E84'.

We then increased the access level to UPDATE and re-ran it.

This time the CSNBKRC succeeded, but the CSNBKRW failed with:

```
ICH408I USER(LENNIE  ) GROUP(SYS1    ) NAME(LENNIE DYMOKE-BRADSH)
  LENNIE.DESKEY.TEST001 CL(CSFKEYS )
  INSUFFICIENT ACCESS AUTHORITY
  ACCESS INTENT(CONTROL)  ACCESS ALLOWED(UPDATE )
```

The request ended with return code 8 and reason code X'3E84'.

We finally increased the access level to CONTROL and re-ran it. This time no messages were produced and the process ran to completion.

Extra notes

Note the following:

- This control is not deemed a key store policy, and hence it does not produce message CSFM608I when activated. Instead it produces message CSFM610I, as noted above.
- No key store policy is required to be active for these key label access levels to work.
- If the key token checking key store policy is active, then it will work with the key label access levels to control access in that same way as if a label had been specified.
- All the above checking has been performed using symmetric keys in the CKDS, but similar controls will work with asymmetric keys in the PKDS.

5.5.7 Symmetric key export controls

This section deals with the need to restrict the exportation of keys under a transport key. This level of access is independent of the access levels described in the previous section. Therefore, it is possible to grant access to export a key token without granting any access to use the key token in an encryption operation.

In addition, it is likely that one would allow use of the token for encryption but disallow the ability to export it.

Exporting of token takes place using the ICSF API CSNDSYX (or CSNFSYX).

Enablement

To make the authority to export a token independent of the hierarchy of access levels, a new RACF class is used called XCSFKEY. Each key that needs controls on who can export it will need to be defined to this new class, using the same syntax as its definition in the CSFKEYS class.

Note: In the introduction of this control no changes are made to the access required to use the asymmetric key used to encrypt the symmetric key. Therefore, READ access to the asymmetric key under the CSFKEYS class is required.

To enable this capability, an active key store policy for the CKDS is required. Therefore, one of the following profiles must exist in the XFACILIT class:

```
CSF.CKDS.TOKEN.CHECK.LABEL.WARN
CSF.CKDS.TOKEN.CHECK.LABEL.FAIL
CSF.CKDS.TOKEN.NODUPLICATES
```

In addition, you must define at least one of the following profiles in the XFACILIT class:

```
CSF.XCSFKEY.ENABLE.AES
CSF.XCSFKEY.ENABLE.DES
```

The commands that might be needed are:

```
RDEFINE XFACILIT CSF.XCSFKEY.ENABLE.AES UACC(N)
RDEFINE XFACILIT CSF.XCSFKEY.ENABLE.DES UACC(N)
SETROPTS RACLIST(XFACILIT) REFRESH
```

Checking the process

We defined the above profiles and also ensured that we had a token check policy active. This caused ICSF to issue the following message:

```
CSFM611I XCSFKEY EXPORT CONTROL FOR AES IS ENABLED.
```

We then defined our test key to the XCSFKEY RACF class:

```
RDEFINE XCSFKEY LENNIE.DESKEY.TEST001 UACC(N)
```

We then attempted to export this key under an RSA key, using CSNDSYX. This was not allowed and produced the following message:

```
ICH408I USER(LENNIE  ) GROUP(SYS1    ) NAME(LENNIE DYMOKE-BRADSH
  LENNIE.DESKEY.TEST001 CL(XCSFKEY )
  INSUFFICIENT ACCESS AUTHORITY
  ACCESS INTENT(UPDATE )  ACCESS ALLOWED(NONE   )
```

The CSNDSYX call ended with return code 8 and reason code X'3E84'.

We then increased the access level to UPDATE, and then the export process worked.

Extra notes

Note the following:

- This control is not deemed a key store policy, and hence it does not produce message CSFM608I when activated.
- This control requires a key policy to be active for the CKDS. A PKDS policy will not enable this control.
- Profiles in the XCSFKEY class can be set to WARNING mode in the same way as profiles in any other class (including CSFKEYS and CSFSERV). This might be achieved as follows:

  ```
  RDEFINE XCSFKEY LENNIE.DESKEY.TEST001 UACC(N) WARNING
  ```

- This is probably a good idea if you are concerned about not having identified all places where the keys are being exported.
- Separate controls are in place for AES keys and DES keys. These can be set entirely differently if required.
- If export controls are not in place for either of the two types of keys, then the ability to export keys of that type will be controlled by access to the CSFKEYS as per normal.

5.5.8 PKA key management extensions control: Part 1

These extensions control the ability to perform export functions. This export function is the service provided by CSNDSYX and CSNFSYX.

Part 1 relates to controlling which asymmetric keys can be used to export symmetric keys, and also how specific asymmetric keys can be used. Part 2 relates to controlling which asymmetric keys can be used to export specific symmetric keys.

These controls are only available with HCR7770 and with z/OS V1R10 and later.

Each of these controls requires the use of the ICSF segment on resource profiles in either or both the CSFKEYS class and the XCSFKEY class.

As a result of the requirement for the ICSF segment, you must have the following in order to use this control:

- ICSF level HCR7770
- z/OS V1 R11 *or* z/OS V1 R10 with the PTF for APAR OA28439

The ICSF segment contains several functions (one of which we have already discussed in “Preventing a secure key from being used as a protected key” on page 64). In this case we need to make use of the ASYMUSAGE parameter.

The rest of this section relates only to part 1 of the PKA key management extensions.

Enablement

To use of this control, several things must be in place. Firstly, a CKDS key store policy must be active and a PKDS key store policy must be active. Once these are in place, at least one of the following profiles must be created:

```
CSF.PKAEXTNS.ENABLE
CSF.PKAEXTNS.ENABLE.WARNONLY
```

Then you must also create the ICSF segment with the ASYMUSAGE parameter on the CSFKEYS or XCSFKEY profiles that protect the key labels that you are going to restrict.

You might want to take a cautious stance and use the WARNONLY mode. You might also want to ensure that you have the ICSF segment information in place before you create the enablement profiles. For example, the commands to create the ICSF segment information could be as follows:

```
RALTER CSFKEYS LENNIE.RSAKEY.TEST001 ICSF(ASYMUSAGE(NOSECUREEXPORT))
SETROPTS RACLIST(CSFKEYS) REFRESH
```

This would prevent the LENNIE.RSAKEY.TEST001 key from being used to export keys.

The commands to enable the controls would be:

```
RDEFINE XFACILIT CSF.PKAEXTNS.ENABLE UACC(N)
SETROPTS RACLIST(XFACILIT) REFRESH
```

The full set of options for the ASYMUSAGE are:

```
SECUREEXPORT | NOSECUREEXPORT
HANDSHAKE | NOHANDSHAKE
```

Checking the process

Our check simply attempted to export a key when we had specified that it should not be allowed to be exported.

We ensured that we had an active key store policy in place for both the CKDS and PKDS and then defined the profiles above. ICSF issued the following message:

```
CSFM612I PKA KEY EXTENSIONS CONTROL IS ENABLED.
```

We then attempted to export a key. The attempt failed with return code 8 and reason code X’0BF5’.

Note that no RACF messages were issued.

We then altered the definition for the key by issuing the following RACF command:

```
RALTER CSFKEYS LENNIE.RSAKEY.TEST001 ICSF(ASYMUSAGE(SECUREEXPORT))
```

```
SETROPTS RACLIST(XFACILIT) REFRESH
```

Following this command, we were able to export the key.

Extra notes

Note the following:

- We did not experiment with the options to allow and disallow the key to be used in handshake operations.
- Note that the ICSF segment can be defined on the key label definition in the XCSFKEY resource class as well as CSFKEYS. This would only be used when you have defined EXPORT controls on the key in question.
- When this control prevents an export from occurring, an SMF record type 82 subtype 27 is written.

5.5.9 PKA key management extensions control: Part 2

Part 2 of the PKA key management extensions is designed to control which public keys can be used to export a symmetric key. The same prerequisites exist for this check as for the previous section:

- ICSF level HCR7770
- z/OS V1 R11 *or* z/OS V1 R10 with the PTF for APAR OA28439

This control also makes use of the ICSF segment, but this time the segment is used on a symmetric key.

Enablement

Just like part 1 of these PKA key management extensions, to use this control, several things must be in place. Firstly, a CKDS key store policy must be active and a PKDS key store policy must be active. Once these are in place, at least one of the following profiles must be created:

```
CSF.PKAEXTNS.ENABLE
CSF.PKAEXTNS.ENABLE.WARNONLY
```

Then you must also create the ICSF segment with the SYMEXPORTABLE parameter on the CSFKEYS or XCSFKEY profiles, which protect the key labels that you are going to restrict. For example, the commands to create the ICSF segment information could be as follows:

```
RALTER CSFKEYS LENNIE.DESKEY.TEST001 ICSF(SYMEXPORTABLE(BYLIST) -
SYMEXPORTKEYS(LENNIE.RSAKEY.TEST001))
SETROPTS RACLIST(CSFKEYS) REFRESH
```

This would allow the LENNIE.DESKEY.TEST001 key to be exported using the key LENNIE.RSAKEY.TEST001.

The commands to enable the PKA key management extensions controls would be:

```
RDEFINE XFACILIT CSF.PKAEXTNS.ENABLE UACC(N)
SETROPTS RACLIST(XFACILIT) REFRESH
```

Checking the process

Our check simply attempts to export a key when we have specified that it should not be allowed to be exported by the key that we specified.

Instead of issuing the exact commands above that we first issued:

```
RALTER CSFKEYS LENNIE.DESKEY.TEST001 ICSF(SYMEXPORTABLE(BYNONE) -
SYMEXPORTKEYS(LENNIE.RSAKEY.TEST001))
SETROPTS RACLIST(CSFKEYS) REFRESH
```

Then we tried the export. We found that the export failed with return code 8 and reason code x'0BF6'.

We then altered the profile using the following command:

```
RALTER CSFKEYS LENNIE.DESKEY.TEST001 ICSF(SYMEXPORTABLE(BYLIST) -
SYMEXPORTKEYS(LENNIE.RSAKEY.TEST001))
SETROPTS RACLIST(CSFKEYS) REFRESH
```

Then we repeated the test. This time the export was successful.

Extra notes

Note the following information:

- We did not experiment with the options to allow and disallow controls by certificates or by using
- The ICSF segment can be defined on the key label definition in the XCSFKEY resource class as well as CSFKEYS. This would be used when you have defined EXPORT controls on the key in question.
- When this control prevents an export from occurring, an SMF record type 82 subtype 27 is written.

5.5.10 Messages

When starting ICSF you will receive a message stating the status of these various key store policy options (Figure 5-4).

```
CSFM607I A CKDS KEY STORE POLICY IS NOT DEFINED.
CSFM607I A PKDS KEY STORE POLICY IS NOT DEFINED.
CSFM610I GRANULAR KEYLABEL ACCESS CONTROL IS DISABLED.
CSFM611I XCSFKEY EXPORT CONTROL FOR AES IS DISABLED.
CSFM611I XCSFKEY EXPORT CONTROL FOR DES IS DISABLED.
CSFM612I PKA KEY EXTENSIONS CONTROL IS DISABLED.
```

Figure 5-4 Messages

Whenever a change in status occurs a message is issued, so it should be possible to work back through the messages and determine the status of each control.

5.6 RACF segment for ICSF

The RACF segment for ICSF was introduced at z/OS Version 1 Release 11, but is available for z/OS Version 1 Release 10 by applying the PTF for APAR OA28439. We documented the use of this segment in the sections above and also in the section on CPACF protected keys.

The complete set of uses for the segment to date follows. In all cases the manual *z/OS Security Server RACF Command Language Reference*, SA22-7687-14, should be treated as the definitive source of information.

5.6.1 ASYMUSAGE

This parameter is explained in “Enablement” on page 100. It is designed to be defined on the RACF resource profile for asymmetric keys, and it specifies certain restrictions a key can be used. The possibilities are:

```
SECUREEXPORT | NOSECUREEXPORT
HANDSHAKE | NOHANDSHAKE
```

5.6.2 SYMEXPORTABLE

This parameter is explained in “Enablement” on page 101. In this case it is designed to be used on the RACF resource profile for symmetric keys. It is used to specify whether the key can be exported under an asymmetric key, and if so by what means the set of asymmetric keys is to be defined. The possibilities are:

```
BYANY | BYLIST | BYNONE
```

This parameter is designed to be used in conjunction with the SYMEXPORTERCERTS and the SYMEXPORTERKEYS parameters, below.

5.6.3 SYMEXPORTCERTS

This parameter is designed to be used in conjunction with the SYMEXPORTABLE parameter. In this case, it is designed to be used on the RACF resource profile for symmetric keys. This parameter can be specified in one of two different formats, depending on whether the trusted identities are specified in a set of PKCS#11 tokens or in a SAF key-ring.

In the case of the PKCS# 11 tokens, cka-id and cert-label should be specified. In the case of SAFkey-ring, the certificate labels are listed in the format user ID and cert-label.

5.6.4 SYMEXPORTKEYS

This parameter is designed to be used in conjunction with the SYMEXPORTABLE parameter. In this case, it is designed to be used on the RACF resource profile for symmetric keys. This parameter can specify a set of ICSF key labels that reference asymmetric keys that can be used to export the symmetric key.

5.6.5 SYMCPACFWRAP

This parameter is used in conjunction with the CPACF protected keys capability. See “Preventing a secure key from being used as a protected key” on page 64 for details.

This keyword is used on a secure symmetric key to specify that is it acceptable to use it as a CPACF protected key.

5.7 RACF resources for ICSF

There is now a great deal of RACF classes that are involved in the use of ICSF services. This section provides a summary of the RACF classes and what they are used for.

5.7.1 CSFSERV

This RACF class is used primarily to protect the APIs that can be used by programmers. The resources that it protects consists of a set of names that start with the characters CSF. Most of these resource names correspond to APIs. For example, the service CSNBENC can be protected by a resource profile named CSFENC. The full set of resource names that are used in ICSF can be found in section 4.3 in the *ICSF Administrators Guide*, SA22-7521-14.

Some of the resources in this list represent APIs that are not documented. For example, some of these resources are used by the ICSF panel application, and access to the resources is required to use that particular function. One instance of this is the resource CSFSMK (set master key) is used in the panel dialogue to set a new master key.

Some of the ICSF APIs cannot be protected in this way, as they do not have resources to protect them. An example of this is the service CSNBSYE, which makes use of CPACF instructions to encipher data using a clear key or a protected key.

The CSFSERV class can make use of generic profiles if you want. For example, a resource of name CSF* can be used to protect all the CSFSERV resources that do not have explicit protection.

This class can be RACLISTed if you want. However, it is not mandatory to do so. If profiles are not defined, then the corresponding resources are not protected. Only READ access is meaningful for this class. If you grant UPDATE or CONTROL access, then this is the same as granting READ access. Note that if you define discrete profiles in this class, ALTER access will allow users to change the access list.

5.7.2 CSFKEYS

This RACF class protects all keys in any of the key stores (that is, CKDS, PKDS, TKDS). Each key has a label that is between 1 and 64 bytes in length.

Resource names for keys are entirely the province of the installation. Many installations have formal naming standards to ensure that no key name clashes. This normally needs to ensure that keys within the different key stores have different names.

Resource profiles can be generic or discrete.

As previously discussed in 5.5.6, “Key label access levels” on page 96, the access levels needed to use a key might depend on how the key is being used.

Note that by default key labels are *not* protected, so you might wish to consider defining a *back-stop* profile such as two asterisks (**).

The CSFKEYS class can be RACLISTed if desired. However, it is not mandatory to do so.

5.7.3 DATASET

The DATASET class should be used to protect all the ICSF keys stores and the parameter files used by ICSF. These will be named according to local standards. The key stores might contain clear keys (especially the TKDS), so you will need to protect these from prying eyes. These data sets should be protected in a similar manner to your RACF databases.

5.7.4 XFACILIT

As seen in previous sections, resources in the XFACILIT class is used by ICSF to control key store policies. The names of all these profiles are shown in section 4.4 in the *ICSF Administrators Guide*, SA22-7521-14.

Note that the XFACILIT class is used by multiple z/OS components, not just ICSF.

5.7.5 XCSFKEYS

The XCSFKEYS class can also be used to define cryptographic key labels. However, resources in this class are only referenced by the symmetric keys label export APIs, CSNDSYX and CSNFSYX. The structure of resource names in the XSCFKEY class is the same as for the CSFKEYS class. See 5.5.7, “Symmetric key export controls” on page 98, for more details.

5.7.6 CRYPTOZ

The CRYPTOZ class is used to define profiles representing PKCS#11 resources. Resource names used for CRYPTOZ are:

`USER.`*`token-name`*
`SO.`*`token-name`*

Each token-name has two profiles protecting it. The access levels for those resources in the CRYPTOZ class have rather different meanings from those normally found.

Note also that there are resources in this class of the form:

`FIPSEXEMPT.token-label`

These are used to specify that resources do not need to be FIPS 140-2 compliant.

For more details see the following manuals:

- *ICSF Administrators Guide*, SA22-7521-14
- *Writing PKCS #11 Applications*, SA23-2231-02
- *RACF Security Administrator’s Guide*, SA22-7683-13

5.8 Secure AES keys

Starting with ICSF level HCR7751, support is available for Secure AES keys, but only on a Crypto Express2 Coprocessor (CEX2C) with z9 EC, z9 BC, z10 EC, and z10 BC or on a Crypto Express3 Coprocessor (CEX3C) with z10 EC and z10 BC.

Note: ICSF releases before HCR7751 do not support secure AES keys and require APAR OA26579 for toleration.

The following services have been updated to support Secure AES keys:

- Symmetric Key Generate Callable Service - CSNDSYG
- Symmetric Key Import Callable Service - CSNDSYI
- Symmetric Key Export Callable Service - CSNDSYX
- Key Record Write - CSNBKRW

- Key Generate Callable Service - CSNBKGN
- Key Token Build Callable Service - CSNBKTB
- Multiple Clear Key Import Callable Service - CSNBCKM
- Multiple Secure Key Import - CSNBSKM
- Symmetric Algorithm Decipher - CSNBSAD and CSNESAD
- Symmetric Algorithm Encipher - CSNBSAE and CSNESAE
- Key Test - CSNBKYT

AES master key

The AES master key (AES-MK) is used to protect AES keys stored in the CKDS. The process with AES-MK works in a similar manner to the DES-MK for DES keys. The AES-MK value can be entered from the ICSF panels. TKE V5.3 or later can be also used to enter the AES-MK value into a CEX2C or CEX3C.

The AES-MK is a 32-byte (256-bit) key that is used to protect keys used on the CEX2C or CEX3C. It is only available on the IBM System z9 Enterprise Class, IBM System z9 Business Class, IBM System z10 Enterprise Class, and IBM System z10 Business Class with a November 2008 or later licensed internal code (LIC).

Creating a new CKDS with AES master key

The first time you start ICSF, you must:

1. Create a cryptographic key data set (CKDS).
2. Enter a new DES-MK into each CEX2C/CEX3C (optional).
3. Enter a new AES-MK into each CEX2C/CEX3C (optional).
4. Initialize the CKDS.

When you initialize the CKDS, ICSF creates a header record for the CKDS and sets any DES or AES master keys in the new master key registers. Next, ICSF sets the DES or AES master key, if any. ICSF then adds the required system key to the CKDS and refreshes the CKDS. When you initialize a CKDS, you can copy the disk copy of the CKDS to create other CKDSs for use on the system.

Note: On the CKDS initialization panel you can improve performance by answering N to Record authentication required.

Updating an existing CKDS with the AES master key

On systems that support the AES master key, you can add the AES master key to any existing CKDS. It is also possible to add the DES master key to a CKDS that was initialized with only the AES master key.

To update the CKDS:

1. Load the new AES master key by using the master key entry panels or by using TKE. The AES master key must be loaded on all active coprocessors.
2. From the Primary Menu, select option 2, MASTER KEY MGMT.
3. Select option 1, INIT/REFRESH/UPDATE CKDS.
4. The Initialize a CKDS panel appears. In the CKDS field, enter the name of an existing initialized CKDS.
5. Choose option 3, Update an existing CKDS, and press Enter. ICSF will check the status of the new master key registers, and the master key verification pattern of the master key is written to the CKDS header record. Note that all the CKDSs that you want to update should be processed prior to going to step 6.

6. In the CKDS field, enter the name of the updated CKDS that will be the active CKDS.
7. Select option 2, REFRESH, and press Enter. The in-storage copy of the CKDS will be updated with your updated CKDS.
8. Return to the Master Key Management panel by pressing END. Choose option 2, SET MK, and press Enter. ICSF sets the AES master key and your system can be used to encrypt AES key operations.

AES master key change

For security reasons, your installation should change the master keys periodically. A DES-MK or AES-MK master key and a CKDS that contains keys that are enciphered under that master key should already exist.

There are three main steps involved in changing the DES-MK master key or AES-MK master key:

1. Enter the DES-MK or AES-MK master key parts.
2. Reencipher all disk copies of the CKDS under the new DES-MK or AES-MK master key.
3. Change the new DES-MK or AES-MK master key and activate the reenciphered CKDS.

Note: When changing the master key, remember to change the name of the CKDS in the installation options data set.

DES and AES master keys can be changed separately or together.

5.9 RSA key lengths

Starting with ICSF level HCR7750, there is new support for up to 4096-bit RSA key length. This 4096-bit RSA key is limited to z9 EC, z9 BC, z10 EC, and z10 BC with CEX2 or CEX3 coprocessors.

Several ICSF APIs have been updated to support the new RSA key length. A list of these APIs is provided in 5.12, “Update on ICSF APIs since HRF7730” on page 111.

Note: The 2048-bit RSA keys can have an exponent in the range of 0 - 256. The 4096-bit RSA key exponents are restricted to the values 3 and 65537.

Migrating to a larger PKDS

To support a larger RSA key length than 2048-bit, the PKDS LRECL size must be increased. This change allows 4096-bit RSA public and private keys to be stored in the PKDS. With ICSF release HCR7750 or later, ICSF expects the PKDS to have the longer LRECL before it will start.

If you currently have a PKDS that ICSF is using and are planning to move to the ICSF HCR7750 or later web deliverable z/OS V1.10 or later, then you need to perform the following steps before starting the new version of ICSF. The steps take you through the tasks that must be performed to make an exact copy of the old PKDS contents. The ICSF PKA services are not available during the time that the copy is made to ensure the PKDS contents are not changing. You need to schedule the change for a period of time when the applications using the PKDS keys are also not available. The steps to migrate are:

1. If the PKDS is shared with down-level systems, install the toleration APAR on those systems to allow continued sharing of the PKDS. The toleration APAR number is OA21807.

2. Create the larger PKDS. Use the JCL in SYS1.SAMPLIB(CSFPKDS) from the HCR7750 or later system. If the PKDS will be shared, place the VSAM data set where it can be shared.
3. Suspend activity with the PKDS. Disable the PKDS READ, PKDS WRITE, PKDS CREATE, and PKDS DELETE access from the ADMINCNTL option. This prevents any updates from being made while the migration action is performed. It affects applications that use the PKDS services.
4. If the old PKDS is not empty, copy it to the larger PKDS using the JCL in SYS1.SAMPLIB(CSFPKDCP) from the HCR7750 system. If the original PKDS is an empty one, you will need to initialize the new PKDS.
5. Protect the VSAM data set from use by non-authorized personnel.
6. Update the ICSF started procedures on all systems to reference the new PKDS.
7. Activate the new PKDS on each system. Refresh the PKDS from the Master Key Mgmt option on the main ICSF Administration panel.
8. Resume activity with the PKDS. Enable the PKDS READ, PKDS WRITE, PKDS CREATE, and PKDS DELETE access from the ADMINCNTL option. Resume any applications that use the PKDS services.

Another approach is to stop ICSF, create the new PKDS, perform the copy, update the installation options data set, and restart ICSF.

5.10 Key stores in a sysplex

As we know, the CKDS is managed on each individual system by having all the key labels and key tokens loaded into a data space. However, unless the sysplex support is enabled, each data space will have their keys managed independently, even though they might share the same CKDS data set and even though the crypto coprocessors for each LPAR have the same master keys.

Sysplex support for updates to the CKDS has been available since HRF7730. This capability enables a single view of the CKDS key store, such that a change to the key store on one system causes a change to the key stores on others systems in the sysplex that share the same CKDS. This change is driven from the ICSF where a key change has been made, to all the other members of the sysplex. This applies to updates that are made via the ICSF APIs (such as CSNBKRC, CSNBKRR, CSNDKRC, and CSNDKRR). Those updates made via CSFKGUP only update the CKDS data set itself, so that the CSFEUTIL program has to be used to drive the updates into the CKDS data space on each system in the LPAR.

These updates are driven from the updating system to all others in the sysplex by making use of the sysplex communications capability called cross-coupling facility (XCF). Sysplex communications links are either via links to a coupling facility or make use of channel-to-channel (CTC) connections between pairs of LPARs within the sysplex.

Note: Details of how the sysplex sharing works can be found in*z9-109 Crypto and TKE V5 Update*, SG24-7123. This supplies sample programs for testing the process of updating keys in a sysplex.

This level of cross-system communication is now available for the other two keys stores, the PKDS and the TKDS.

5.10.1 Background information about the PKDS and TKDS

Prior to HCR7751, the PKDS was managed using a local cache. This cache had a default size of 64 records, and a maximum size of 128 records. However, if the PKDS data set was shared among members of a sysplex, then an update to the PKDS on one system would not be apparent on another system until a refresh was performed using either the batch program CSFEUTIL or the ISPF panel option 2.7.

An alternative to this was to set the value of the cache to zero. To do this the options data set had to specify:

```
PKDSCACHE(0)
```

This meant that all the speed benefits of having a cache were lost.

The PKDS management was changed in HCR7751 in two ways:

- The PKDS was loaded into a dataspace similarly to the CKDS.
- The PKDS was enabled for sysplex sharing.

The TKDS was introduced at ICSF level HCR7740 and had the data space and sysplex updating capability built into its support at the beginning.

5.10.2 How to enable Sysplex sharing

The main requirement for sharing the PKDS or the CKDS is that the systems make use of identical master keys, corresponding to the key store.

This requirement does not exist for the TKDS, which does not have a corresponding master key. (Keys in the TKDS are not encrypted.)

Let us assume that we have two or more systems in a sysplex and we have a single instance each of the CKDS, PKDS, and TKDS. In this case, we can specify the following options in the ICSF options data set,

```
SYSPLEXCKDS(YES,FAIL(YES))
SYSPLEXPKDS(YES,FAIL(YES))
SYSPLEXTKDS(YES,FAIL(YES))
```

Note: In each case above, the FAIL option could have been set to NO. This means that if it was not possible to establish sysplex communications (that is, join the sysplex group SYSICSF), then initialization of ICSF would continue.

When the above options are specified, this causes ICSF to issue XCF broadcast messages to other systems in the sysplex whenever a key in each of the data sets is changed. The message indicates to the other systems that they should re-read the appropriate record from the appropriate key store.

If the first parameter for any of the above keywords is set to NO, then that system will not participate in the sysplex sharing and will continue to process as normal.

If any of the SYSPLEX keywords are omitted, then the sharing capability for that key store is deemed to be NO (that is, no sysplex consistency sharing).

5.10.3 Restrictions on PKDS sharing

If a PKDS is shared between a z800 or z900 and other systems, then we suggest that the CCF on the z800 or z900 have the SMK and the KMMK set to be equal. If this is not done, then the DSA and RSA token encrypted under the KMMK will not be useable on the other systems.

5.10.4 Changing the PKA master key when the PKDS is shared in a sysplex

The process for changing the asymmetric (PKA) master key when the PKDS is shared in a sysplex is the same as if the PKDS is shared at all.

Let us assume that we have three systems, A, B, and C. We also have a new PKDS data set defined. If you have not defined the new PKDS, then you will need to create one large enough for all the keys that you plan to use. Sample JCL can be found in SYS1.SAMPLIB(CSFPKDS).

We decide to encipher the PKDS on system A, so we proceed as follows:

1. On system A, from the ICSF main panel choose option 4 (ADMINCNTL).
2. On the ADMINCNTL panel enter a `D` beside the PKA Callable Services option and press Enter. The setting should change to show that it is disabled. The setting for Dynamic PKDS Access will also show as disabled.
3. On system B and on system C, select the same ADMINCNTL panel.
4. On system B and system C enter a `D` beside Dynamic PKDS Access and press Enter.
5. On system A change the master key. To do this you will need to:
   a. Select option 1 from the ICSF main panel.
   b. Place an `E` beside all the co-processors whose key is to change and press Enter. The panel will change to the Master Key Entry panel.
   c. Enter the new master key. (This can be done in the same manner as entering a key for a non-shared system. See the *ICSF Administrator's Guide*, SA22-7521-14, for more details.)
6. Re-encipher the old PKDS data set and create a new PKDS data set. To do this you will need to do the following:
   a. Select ICSF option 2 (Master Key Management).
   b. Select option 6 (Reencipher PKDS).
   c. Enter the names of the old and new PKDS data sets.
   d. Press Enter to reencipher the data set.
7. Press Return to go back to the Master Key Management panel.
8. Select option 7 (Refresh PKDS) and enter the name of the new PKDS data set.
9. Press return twice to go back to the main ICSF panel, then select option 4 (ADMINCNTL).
10. Enter `E` for Dynamic PKDS Access and PKA Callable Services, and press Enter. Each should now be enabled.
11. Move to system B and change the master key as you did in step 5 above for system A.
12. Still on system B, select the ADMINCNTL panel and enter `E` for Dynamic PKDS Access and PKA Callable Services, and then press Enter. Each should now be enabled.
13. Repeat steps 11 and 12 for system C.

Note: Before you have finished, remember to alter the name of the PKDS data set in the PKDSN parameter in the parmlib member for each of your ICSF started tasks. If you do not do this, then ICSF starts with the old PKDS data set and causes problems.

5.10.5 PKDSCACHE

The PKDSCACHE keyword is no longer supported. However, it is tolerated, and will produce message CSFO0212 and then be ignored.

Instead of using a cache, ICSF now loads the entire PKDS into a dataspace, in a similar manner to the handling of the CKDS and TKDS. This was introduced at level HCR7751.

5.10.6 CSFPUTIL

As an alternative to using the ICSF panel, the CSFPUTIL program can be used to:

- Reencipher a PKDS data set.
- Refresh the in-storage PKDS cache.

JCL to reencipher a PKDS is:

```
//RECIPHER EXEC PGM=CSFPUTIL,PARM='old.PKDS,new.PKDS,RECIPHER'
```

JCL to refresh a PKDS is:

```
//REFRESH EXEC PGM=CSFPUTIL,PARM='REFRESH,new.PKDS'
```

Or simply the following if the PKDS data set has not changed:

```
//REFRESH EXEC PGM=CSFPUTIL,PARM='REFRESH'
```

5.11 PAN lengths

Prior ICSF level HCR7751, the PAN lengths 13, 16, and 19 were the only ones supported by VISA CVV generate or verify services. Those missing length values were updated in ICSF service APIs to generate and verify VISA CVV.

Support has bee added for PAN-14, PAN-15, PAN-17, and PAN-18 on ICSF level HCR7751 and later:

- CSNBCSG - VISA CVV Service Generate
- CSNBCSV - VISA CVV Service Verify

5.12 Update on ICSF APIs since HRF7730

This section shows the changes to the ICSF APIs since the previous IBM Redbooks publication, *z9-109 Crypto and TKE V5 Update*, SG24-7123, which was then based on the ICSF services as provided by ICSF FMID HCR7730. The changes are listed from the most recent ICSF release (HCR7770) to the release that directly followed FMID HCR7730 (that is HCR7731).

5.12.1 ICSF FMID HCR7770

HCR7770 introduced the following changes:

- Added new callable service PKA Key Translate (CSNDPKT and CSNFPKT). Using this callable service, applications can translate a source CCA RSA key token into a target external smart card key token.
- Added new callable services for managing PKCS #11 tokens and objects. These additional services are:
  - PKCS #11 Derive key (CSFPDVK)
  - PKCS #11 Derive multiple keys (CSFPDMK)
  - PKCS #11 Generate HMAC (CSFPHMG)
  - PKCS #11 Generate key pair (CSFPGKP)
  - PKCS #11 Generate secret key (CSFPGSK)
  - PKCS #11 One-way hash generate (CSFPOWH)
  - PKCS #11 Private key sign (CSFPPKS)
  - PKCS #11 Pseudo-random function (CSFPPRF)
  - PKCS #11 Public key verify (CSFPPKV)
  - PKCS #11 Secret key decrypt (CSFPSKD)
  - PKCS #11 Secret key encrypt (CSFPSKE)
  - PKCS #11 Unwrap key (CSFPUWK)
  - PKCS #11 Verify HMAC (CSFPHMV)
  - PKCS #11 Wrap key (CSFPWPK)
- The Symmetric Key Export and Symmetric Key Import callable services now support invocation in AMODE(64).
- The Symmetric Key Encipher (for example, CSNBSYE and CSNBSAE) and Symmetric Key Decipher (for example, CSNBSYD and CSNBSAD) callable services now support an encrypted key in the CKDS. This enables applications to leverage the performance capabilities of CPACF when enciphering/deciphering data using encrypted symmetric keys.
- The ICSF Query Algorithm (CSFIQA and CSFIQA6) utility and ICSF Query Facility (CSFIQF and CSFIQF6) are updated to return additional data.

5.12.2 ICSF FMID HCR7751

HCR7751 introduced the following changes:

- Added support for secure key AES:
  - Added new callable service (CSNBSAD, CSNBSAD1, CSNESAD, and CSNESAD1): Symmetric Algorithm Decipher
  - Added new callable service (CSNBSAE, CSNBSAE1, CSNESAE, and CSNESAE1): Symmetric Algorithm Encipher
- Added support for PAN-14, PAN-15, PAN-17, and PAN-18:
  - CSNBCSG - VISA CVV Service Generate
  - CSNBCSV - VISA CVV Service Verify
- Added support for IPv6:
  - Added new callable service (CSNBSMG, CSNBSMG1, CSNESMG, and CSNESMG1): Symmetric MAC Generate
  - Added new callable service (CSNBSMV, CSNBSMV1, CSNESMV, and CSNESMV1): Symmetric MAC Verify

- Added new callable service (CSFIQA and CSFIQA6): ICSF Query Algorithm.
- Symmetric keys can now refer to DES, TDES, or AES keys.
- These callable services have been changed to support secure key AES:
  - CSFIQF and CSFIQF6 - ICSF Query Function
  - CSNBKGN and CSNEKGN - Key Generate
  - CSNBKRC - Key Record Create
  - CSNBKRD - Key Record Delete
  - CSNBKRR - Key Record Read
  - CSNBKRW - Key Record Write
  - CSNBKYT - Key Test
  - CSNBKTX - Key Test Extended
  - CSNBKTB - Key Token Build
  - CSNBCKM and CSNECKM - Multiple Clear Key Import
  - CSNBSKM - Multiple Secure Key Import
  - CSNDSYX - Symmetric Key Export
  - CSNDSYG - Symmetric Key Generate
  - CSNDSYI - Symmetric Key Import

5.12.3 ICSF FMID HCR7750

HCR7750 introduced the following changes:

- CPACF supports the AES algorithm for 192-bit and 256-bit keys for CSNBSYE, CSNBSYD, CSNBSYE1, and CSNBSYD1.
- These callable services have been changed to support new SHA-2 algorithms:
  - CSNBOWH, CSNBOWH1, CSNEOWH, and CSNEOWH1 - One-Way Hash Generate
  - CSFIQF and CSFIQF6 - ICSF Query Service
- Added support for 4096-bit RSA keys. These callable services have been changed:
  - CSNDPKD and CSNFPKD - PKA Decrypt
  - CSNDPKE and CSNFPKE - PKA Encrypt
  - CSNDPKX and CSNFPKX - PKA Key Extract
  - CSNDPKG and CSNFPKG - PKA Key Generate
  - CSNDPKI and CSNFPKI - PKA Key Import
  - CSNDPKB and CSNFPKB - PKA Key Token Build
  - CSNDKTC - PKA Key Token Change
  - CSNDKRC and CSNFKRC - PKDS Record Create
  - CSNDKRW - PKDS Record Write
  - CSNDKRD and CSNFKRD - PKDS Record Delete
  - CSNDRKX - Remote Key Export
  - CSNDSYX - Symmetric Key Export
  - CSNDSYG - Symmetric Key Generate
  - CSNDSYI - Symmetric Key Import
  - CSNDTBC - Trusted Block Create
  - CSNDDSG - Digital Signature Generate
  - CSNDDSV - Digital Signature Verify

- Added support for ISO-3 PIN block format. These callable services have been changed:
  - CSNBCPE - Clear PIN Encrypt
  - CSNBEPG - Encrypted PIN Generate
  - CSNBPTR - Encrypted PIN Translate
  - CSNBPVR - Encrypted PIN Verify
  - CSNBPCU - PIN Change/Unblock
  - CSNBSPN - Secure Messaging for PINs
- The callable service Random Number Generate Long (CSNBRNGL and CSNERNGL) now allows a user to specify the length when generating a random number.

5.12.4 ICSF FMID HCR7740

HCR7740 introduced the following changes:

- These callable services have been added to support PKCS #11 token management:
  - CSFPTRC - Token Record Create
  - CSFPTRD - Token Record Delete
  - CSFPTRL - Token Record List
  - CSFPSAV - Set Attribute Value
  - CSFPGAV - Get Attribute Value
- These callable services have been changed to support Cipher Feedback Mode (CFB) and PKCS #7 padding for encryption:
  - CSNBSYD - Symmetric Key Decipher (new CFB and PKCS-PAD keywords)
  - CSNBSYE - Symmetric Key Encipher (new CFB and PKCS-PAD keywords)
- CSNBCSV/CSNBCSG: Added that on a IBM Eserver zSeries® 800, IBM Eserver zSeries 900 or later, the user must pad out the CVV with blanks if the supplied CVV is less than five characters.

5.12.5 ICSF,FMID HCR7731

HCR7731 introduced the following changes:

- The following callable services have been added to support remote key load:
  - CSNDTBC - Trusted Block Create
  - CSNDRKX - Remote Key Export
- The following callable services have been changed to support remote key load:
  - CSNDPKI - PKA Key Import
  - CSNDDSV - Digital Signature Verify
  - CSNDKTC - PKA Key Token Change

- Added Enhanced PIN Security:
  - Encrypted PIN Translate enhanced further to produce return code 8,3016. The value of the PAD data is not valid.
  - The following are changed:
    - Clear PIN Encrypt - CSNBCPE
    - Clear PIN Generate Alternate - CSNBCPA
    - Encrypted PIN Generate - CSNBEPG
    - Encrypted PIN Translate - CSNBPTR
    - Encrypted PIN Verify - CSNBPVR
    - PIN Change/Unblock - CSNBPCU
- Added support for ISO 16609 CBC mode T-DES. New rule array keyword, TDES-MAC, will be added to the MAC Generate and MAC Verify Services.

5.13 Auditing records

The SMF records below are used to report on hardware cryptography activity in z/OS. Further details on these SMF records and how to collect them can be found in *z/OS MVS System Management Facilities (SMF)*, SA22-7630.

5.13.1 Type 82 (ICSF record)

This SMF record is mainly used to report ICSF administrative and environmental events, except for subtypes 17, 19, and 20, which are intended to provide performance data. The record subtypes are:

- Subtype 1 is written whenever ICSF is started.
- Subtype 3 is written whenever there is a change in the number of available processors with the cryptographic feature. (This subtype relates to previous coprocessors technology now obsoleted beginning with System z990.)
- Subtype 4 is written whenever ICSF handles error conditions for cryptographic feature failure (CC3, Reason Code 1) or cryptographic tampering (CC3 Reason Code 3).
- Subtype 5 is written whenever a change to special security mode is detected.
- Subtypes 6 and 7 are written whenever a key part is entered via the key entry unit (KEU). (This subtype relates to previous coprocessor technology now obsoleted.)
- Subtype 8 is written whenever the in-storage copy of the CKDS is refreshed.
- Subtype 9 is written whenever the CKDS is updated by a dynamic CKDS update service.
- Subtype 10 is written when a clear key part is entered for one of the PKA master keys.
- Subtype 11 is written when a clear key part is entered for the DES master key.
- Subtype 12 is written for each request and reply from calls to the CSFSPKSC service by TKE. (This subtype relates to previous coprocessor technology now obsoleted beginning with System z990.)
- Subtype 13 is written whenever the PKDS is updated by a dynamic PKDS update service.
- Subtype 14 is written when a clear key part is entered for any of the Crypto Express2/Express3 Coprocessor master keys.
- Subtype 15 is written whenever a Crypto Express2/Express3 Coprocessor retained key is created or deleted.

- Subtype 16 is written for each request and reply from calls to the CSFPCI service by TKE.
- Subtype 17 is written periodically to provide an indication of Crypto Express2/Express3 Coprocessor usage.
- Subtype 18 is written when a Crypto Express2/Express3 Coprocessor or Crypto Express2/Express3 Accelerator comes online or goes offline.
- Subtype 19 is written when a Crypto Express2/Express3 Cryptographic Coprocessor operation begins or ends.
- Subtype 20 is written by ICSF to record processing times for Crypto Express2/Express3 coprocessors.
- Subtype 21 is written when ICSF issues IXCJOIN to join the ICSF sysplex group or issues IXCLEAVE to leave the sysplex group.
- Subtype 22 is written when the Trusted Block Create Callable services are invoked.
- Subtype 23 is written when the token data set (TKDS) is updated.
- Subtype 24 is written whenever a duplicate token is found.
- Subtype 25 is written for a key store policy violation.
- Subtype 26 is written whenever the in-storage copy of the PKDS is refreshed.
- Subtype 27 is written for PKA Key Management Extensions violation.
- Subtype 28 is written to report CPACF protected key related events.

Note: A sample job is supplied in z/OS to format the ICSF type 82 SMF records and gives a report of ICSF activity. The job is available in SYS1.SAMPLIB(CSFSMFJ) and invokes a REXX exec SYS1.SAMPLIB(CSFSMFR).

IBM eServer zSeries 990 (z990) Cryptography Implementation, SG24-7070, provides examples of the use of these utilities.

5.13.2 Type 70 - Subtype 2 (RMF Processor Activity)

This record contains measurement data for cryptographic coprocessors and accelerators located in the following sections:

- The Cryptographic Coprocessor Data section, which contains measurement data for cryptographic coprocessors, as provided by the Crypto Express2/Express3 features for the coprocessors that they contain.
- The Cryptographic Accelerator Data section, which contains measurement data for cryptographic accelerators, as provided by the Crypto Express2/Express3 features for the coprocessors in accelerator mode that they contain.
- The ICSF Services Data section, which contains measurement data for the set of Integrated Cryptographic Service Facility (ICSF) services that have been selected, by ICSF design, to be reported by RMF.

This record is written for each measurement interval and when the session terminates.

5.13.3 Type 30 (Common Address Space Work)

This record contains the field SMF30CSC, which is described as the “Integrated Cryptographic Service Facility/MVS (ICSF/MVS) service count.” This is the number of cryptographic instructions executed on behalf of the caller. Each time that ICSF issues a

command to a hardware coprocessor, this count is incremented by one. However, this means that in some cases, like bulk encryption of data, the count would be incremented several times, as ICSF might loop on commands being issued to the coprocessor although performing a single service call at the ICSF API level. For other operations, like a PIN verification, the count would be incremented by one for a single service call to ICSF. There is therefore no guaranteed correlation between the number of cryptographic instructions and the actual number of service calls to ICSF.

5.13.4 Type 72 - Subtype 3 (Workload Activity)

Two fields are used to track, by sampling, the status of tasks waiting to get access to an available hardware cryptographic coprocessor (or adjunct processor):

- R723APUAP, which is the count of *crypto using* samples (that is, a TCB was found executing on a cryptographic coprocessor)
- R723APD, which is the count of AP crypto delay samples (that is, a TCB was found waiting for a cryptographic coprocessor)

These counters directly reflect, on a sampling basis, the number of the requests being queued by ICSF, because they exceed the ICSF-assigned threshold for requests waiting for service and the number of requests that are currently being executed by the coprocessor.

5.14 64-bit services

ICSF provides 64-bit addressing mode support to a subset only of the callable services. These services can still be called in their 31-bit addressing mode version or called in their 64-bit addressing mode version. For the latter, the name of the service is changed in that the prefix part of the name reads:

- CSNExxx for services named CSNBxxx in their 31-bit addressing mode version
- CSNFxxx for services named CSNDxxx
- CSF6xxx for services named CSFxxx
- CSF1xxx for services named CSFPxxx

Applications that are written for AMODE(64) operation must be linked with the ICSF 64-bit service stubs and must invoke the service with the appropriate service name.

Customer installation exits that were written for the 31-bit addressing mode version of these services must be rewritten to ensure that the exits can handle AMODE(64) operation.

These services support invocation in AMODE(64):

- Clear Key Import - CSNECKI
- Decipher - CSNEDEC
- Digital Signature Generate - CSNFDSG
- Digital Signature Verify - CSNFDSV
- Encipher - CSNEENC
- ICSF Query Algorithm - CSFIQA6
- ICSF Query Facility - CSFIQF6
- Key Generate - CSNEKGN
- Multiple Clear Key Import - CSNECKM
- One Way Hash - CSNEOWH
- PKA Decrypt - CSNFPKD
- PKA Encrypt - CSNFPKE
- PKA Key Generate - CSNFPKG

- PKA Key Import - CSNFPKI
- PKA Key Token Build - CSNFPKB
- PKA Public Key Extract - CSNFPKX
- PKCS #11 Derive key - CSF1DVK
- PKCS #11 Derive multiple keys - CSF1DMK
- PKCS #11 Generate HMAC - CSF1HMG
- PKCS #11 Generate key pair - CSF1GKP
- PKCS #11 Generate secret key - CSF1GSK
- PKCS #11 Get attribute value - CSF1GAV
- PKCS #11 One-way hash generate - CSF1OWH
- PKCS #11 Private key sign - CSF1PKS
- PKCS #11 Pseudo-random function - CSF1PRF
- PKCS #11 Public key verify - CSF1PKV
- PKCS #11 Secret key decrypt - CSF1SKD
- PKCS #11 Secret key encrypt - CSF1SKE
- PKCS #11 Set attribute value - CSF1SAV
- PKCS #11 token record create - CSF1TRC
- PKCS #11 token record delete - CSF1TRD
- PKCS #11 token record list - CSF1TRL
- PKCS #11 Unwrap key - CSF1UWK
- PKCS #11 Verify HMAC - CSF1HMV
- PKCS #11 Wrap key - CSF1WPK
- PKDS Record Create - CSNFKRC
- PKDS Record Delete - CSNFKRD
- Random Number Generate - CSNERNG
- Random Number Generate Long - CSNBRNGL
- Retained Key Delete - CSNFRKD
- Retained Key List - CSNFRKL
- Symmetric Algorithm Decipher - CSNESAD and CSNESAD1
- Symmetric Algorithm Encipher - CSNESAE and CSNESAE1
- Symmetric Key Export - CSNFSYX
- Symmetric Key Import - CSNFSYI
- Symmetric Key Decipher - CSNESYD
- Symmetric Key Encipher - CSNESYE
- Symmetric MAC Generate - CSNESMG and CSNESMG1
- Symmetric MAC Verify - CSNESMV and CSNESMV1

5.15 ICSF query services

The CSFIQF query service has been enhanced in many ways over recent releases. It is now possible to query the status of the ICSF key store policy controls, as detailed in 5.5, “Key store policies” on page 87.

An example of how CSFIQF can be used in REXX is shown in routine REXIQF in Appendix A, “Sample programs in assembler and REXX” on page 261. This program shows the status of many of the ICSF services and also the status of the key store policy controls.

It is also possible to query the status of individual coprocessors using other options in CSFIQF.

New in HCR7751 is the query service CSFIQA. This can be used to determine what cryptographic algorithms are available, what are the maximum lengths they can operate at, and whether the services are supplied by software, coprocessor, or CPU (that is, CPACF).

An example of how to use REXX to do this is shown in routine REXIQA in Appendix A, “Sample programs in assembler and REXX” on page 261.

5.16 The z/OS Health Checker and ICSF

The z/OS Health Checker is a continuously running started task that executes checks programs developed by IBM, independent software vendors, or users, according to the framework and coding rules explained in *IBM Health Checker for z/OS User’s Guide*, SA22-7994.

The z/OS Health Checker and the IBM checks are delivered in z/OS beginning with z/OS V1R7 and are intended to detect common configuration and setups errors that affect many z/OS components, including ICSF, and to issue warning messages accordingly.

The checks execution specifications include an execution interval (one-time check or periodic one) and a default severity (low, medium, or high).

There are currently two checks provided by IBM for ICSF.

ICSFMIG7731_ICSF_RETAINED_RSAKEY

This check detects the existence of retained RSA private keys on a Crypto Express2/Express3 cryptographic card. It is provided beginning with z/OS V1R9. The check is specified as one-time check with low severity.

The user should run the check periodically, when the events occur that affect check results. For example, run the check dynamically when:

- The ICSF product release level is being upgraded to any new ICSF release level.
- The z/OS product release level is being upgraded and ICSF is an exploited feature for that z/OS image.

Reason for check

A Crypto Express2/Express3 card might possess the only copy of a retained RSA private key. Customers that run applications and middleware that utilize the retained key functionality of these cards are exposed to the loss of keys upon hardware failure, which might result from a problem as simple as an exhausted or malfunctioning card battery. Lost retained keys have the further implication of lost data, for retained key management keys, and an inability to verify signatures, for retained signature keys.

Starting with ICSF FMID HCR7750, the user no longer has the ability to store new private RSA keys intended for key management usage in a cryptographic coprocessor. Existing applications will continue to be able to use the retained keys and to delete them from the cryptographic coprocessor cards.

ICSFMIG7731_ICSF_PKDS_TO_4096BIT

The check verifies that the PKDS size in an ICSF pre-HCR7750 environment is sufficiently allocated to support 4096-bit RSA keys. It is provided beginning with z/OS V1R8. The check is specified as a one-time check with low severity.

The user should run the check periodically when events occur that affect check results. For example, run the check dynamically when:

- The ICSF product release level is being upgraded to HCR7750 or later.

- The z/OS product release level is being upgraded and ICSF is an exploited feature for that z/OS image.

Reason for check

ICSF FMID HCR7750 introduces support for 4096-bit RSA keys, which requires a larger PKDS than prior ICSF releases needed. If a customer at a pre-HCR7750 FMID ICSF level migrates to HCR7750 without first reallocating the PKDS for 4096-bit key support, ICSF at HCR7750 fails to start.

This ICSF migration check will detect cases where the currently active PKDS is not sufficiently allocated for the HCR7750 environment and inform the customer that a PKDS reallocation action is necessary.

5.17 UDX

There are three pieces to the User Defined Extension (UDX) functional implementation explained at 2.6.3, “The UDX functional implementation” on page 44. This includes a specific ICSF callable service, which when called by a user application will invoke the UDX code to be executed in the Crypto Express3/Express2 coprocessor. This is a user-defined host service that matches the specific UDX code running in the coprocessor card. Applications are calling the service using a designated service number. Refer to 2.6.3, “The UDX functional implementation” on page 44, for further explanation.

6

z/OS support for the PKCS#11 API

The PKCS#11 cryptographic API support in z/OS has been introduced in z/OS V1R9 with ICSF HCR7740 to provide an alternative to the IBM Common Cryptographic Architecture API in order to broaden the scope of cryptographic applications that can make use of System z integrated hardware cryptography.

This chapter explains the PKCS#11 API implementation and operations as of z/OS V1R11 with ICSF HCR7770. This includes the following improvements to the initial implementation:

- New ICSF callable services have been added so that non-LE programs can invoke the PKCS#11 cryptographic functions.

  Note: This includes support for many encryption and data integrity algorithms, including Elliptic Curve Cryptography (ECC) based algorithms. ECC-based algorithms can use shorter keys and perform faster than RSA-based algorithms.

- All services can run without any cryptographic hardware being available. However, they exploit whatever cryptographic hardware happens to be available.
- ICSF provides the PKCS#11 services even when a TKDS is not defined. However, objects will not persist across sessions in this case.
- A FIPS 140-2 level 1 compliant version of the ICSF PKCS#11 services can be optionally used. Full compliance requires RACF to verify the IBM digital signature of the involved software modules. Program digital signature verification by RACF is available beginning with z/OS V1R11.

This chapter covers these improvements, along with examples of use of the z/OS built-in PKCS#11 token management facilities (that is, the ICSF Token Browser and the extensions to the RACF RACDCERT command and to the gskkyman z/OS UNIX® utility).

Important: Practical use of the ICSF PKCS#11 services also relies on a thorough knowledge of the PKCS #11 v2.20: Cryptographic Token Interface Standard. The standard is published by RSA Laboratories at:

http://www.rsasecurity.com/rsalabs/

Click the **Standards Initiatives** tab.

6.1 Public Key Cryptography Standard #11 (PKCS#11)

RSA Laboratories of RSA Security, Inc., offers its Public Key Cryptography Standards (PKCS) to developers of computer programs that use public key and related technology. PKCS#11, also known as *Cryptoki*, is the cryptographic token interface standard. It specifies an API to devices, referred to as *tokens*, that hold cryptographic information and perform cryptographic functions.

The PKCS#11 API is an industry-accepted standard commonly used by cryptographic applications over the world.

6.1.1 An overview of the PKCS#11 concepts

On most single-user systems, a PKCS#11 token is a smart card or other plug-installable cryptographic device, accessed through a card reader or *slot*. The PKCS#11 specification assigns numbers to slots, known as *slot IDs*. An application identifies the token that it wants to access by specifying the appropriate slot ID. On systems that have multiple slots, the application determines which slot to access.

PKCS#11 tokens are typically created in a factory and initialized either before they are installed or upon their first use. Because PKCS#11 tokens are usually physical hardware devices, the PKCS#11 specification provides no mechanism for deleting tokens.

The PKCS#11 logical view of a token is therefore a device that stores objects and can perform cryptographic functions. PKCS#11 defines five classes of objects:

- A data object that is defined by an application
- A certificate object that stores a certificate
- A public key object that stores an asymmetric public key
- A private key object that stores an asymmetric private key
- A secret key object that stores a secret symmetric key
- A domain parameters object that stores elliptic curve parameters.

Objects are also classified according to their lifetime and visibility. *Token objects* are visible to all applications connected to the token that have sufficient permission, and remain on the token even after the sessions (that is, connections between an application and the token) are closed and the token is removed from its slot. *Session objects* are more temporary, and when a session is closed by any means, all session objects created by that session are automatically destroyed. Furthermore, session objects are visible only to the application that created them.

Attributes are characteristics that distinguish an instance of an object. General attributes in PKCS#11 distinguish, for example, whether the object is public or private. Other attributes are specific to a particular type of object, such as a modules or exponent for RSA keys.

The PKCS#11 standard was designed for systems that grant access to token information based on a PIN code. The standard recognizes two types of access levels:

- Security officer (SO)
- Standard user (USER)

The role of the SO is to initialize a token (zeroize the content) and set the user's PIN. The SO can also access public objects on the token, but not private ones. The user can access private objects on the token. Access is granted only after the user has been authenticated using the PIN code. Users can also change their own PINs. Users cannot, however, re-initialize a token.

6.1.2 The benefits of implementing PKCS#11 on z/OS

z/OS with ICSF support of PKCS#11 provides an alternative to the IBM Common Cryptographic Architecture (CCA) API and broadens the scope of cryptographic applications that can make use of System z cryptography. PKCS#11 applications developed for other platforms can be recompiled and run on z/OS. PKCS#11 tokens are potentially common keystores among applications, including Java applications.

With minor exceptions, RACF and System SSL use different keystores (RACF keyrings and System SSL key databases). Implementing the PKCS#11 token concept in z/OS also offers a common means of storing keys and certificates, which both RACF and System SSL can use.

6.1.3 Mapping PKCS#11 concepts to z/OS cryptographic services implementation

On z/OS PKCS#11 tokens are not cryptographic devices but rather virtual smart cards. New tokens can be created at any time. The tokens can be application specific or system-wide, depending on access control definitions, which are used instead of PINs. Because z/OS PKCS#11 tokens are virtual, z/OS also provides a way to delete them.

In z/OS PKCS#11, which is a new subcomponent of ICSF, tokens and their contents are stored in a dedicated VSAM data set, the Token Key Data Set (TKDS). TKDS serves as the repository for persistent cryptographic objects used by PKCS#11 applications.

Note: In addition to any tokens that your installation might create, ICSF HCR7770 also creates a token that is available to all applications. This *omnipresent* token is created by ICSF to enable PKCS #11 services when no other token has been created. This token supports session objects only (that is, objects that do not persist beyond the life of a PKCS #11 session).

The omnipresent token is always mapped to slot ID #0, and its token label is SYSTOK-SESSION-ONLY.

z//OS provides several facilities to manage PKCS#11tokens:

- A C language application interface (API) that implements a subset of the PKCS#11 specification.
- PKCS#11 ICSF callable services, which are also used by the C API.
- The ICSF ISPF panels, called *token browser*, which provide the capability to see a formatted view of TKDS objects and make minor, limited updates to them. However, this does not apply to objects in the omnipresent token, as they cannot be see from the token browser because of their non-persistence.
- The RACF RACDCERT command has specific functions dedicated to the management of tokens and the certificates and keys that they might contain.
- The gskkyman utility also providing for the management of tokens and keys that they contain.

ICSF supports PKCS#11 session objects and token objects. A session object exists for the life of a PKCS#11 session. ICSF creates a session object data space to hold session objects. An application has only one session object data space, even if the application spawns multiple PKCS#11 sessions. Token objects are stored in the TKDS with one record per object. They are visible to all applications having sufficient permission to the token contents.

The objects are persistent and remain associated with the token even after a session is closed.

The PKCS#11 standard was designed for systems that grant access to token information based on a PIN. PINs are not used on z/OS. Instead, profiles in the SAF CRYPTOZ class of resources control access to tokens. Each token has two resources in the CRYPTOZ class:

- The resource USER.*token-name* controls the access of the user role to the token.
- The resource SO.*token-name* controls the access of the SO role to the token.

The access level to each of these resources (READ, UPDATE, and CONTROL) determines the user's access level to the token and the objects that it contains.

6.2 z/OS PKCS#11 infrastructure and setup

Figure 6-1 shows the infrastructure of the PKCS#11 implementation in z/OS V1R11 with ICSF HCR7770. One main component is the token key data set. However, beginning with ICSF HCR7770, if the TKDS is not available, the PKCS#11 application can still use the *omnipresent* token (to the extent that the omnipresent token is not to store persistent objects). Everything below the dashed line in Figure 6-1 is operating in the ICSF address space. Above the dashed line, operations are performed in the user's own address space.

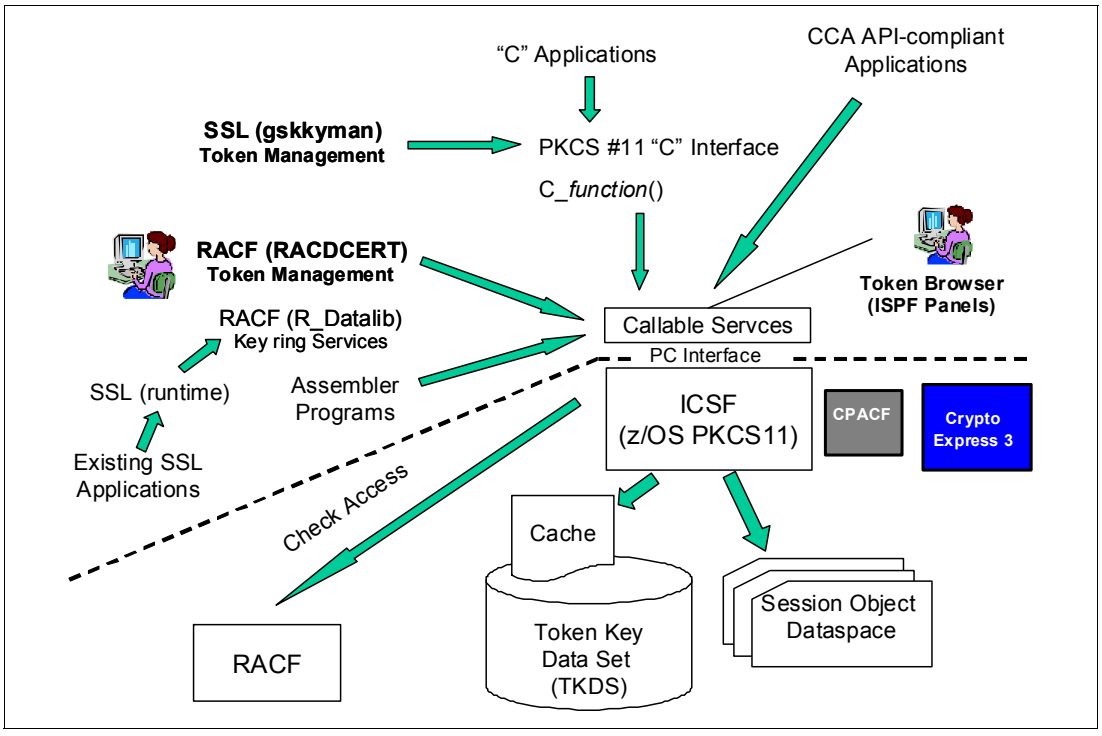


Figure 6-1 Architectural view of z/OS PKCS#11 support

The user interactive interfaces are the token browser panels, the RACF RACDCERT command, and the z/OS System SSL gskkyman utility. Other interfaces are programming interfaces at the ICSF callable services level and at the z/OS C functions level. Note that the interface used by programs to access certificates stored in RACF key rings (the R_datalib RACF callable service) has been extended to give access to z/OS PKCS#11 tokens as well.

6.2.1 Tokens

The PKCS#11 tokens on z/OS are virtual, conceptually similar to RACF (SAF) key rings. An application can have one or more z/OS PKCS#11 tokens, depending on its need. z/OS PKCS#11 tokens are created using the z/OS administrative interfaces, such as those provided by RACF, the gskkyman utility, or applications using the C API or the ICSF callable services. ICSF ISPF panels can also be used for token management with limited usability. See “ICSF panels, token browser” on page 133 for more information.

Each token has a unique token name or label that is specified by the user or application when the token is created. The following rules apply to token names:

- Names can be up to 32 characters in length.
- Permitted characters are:
  - Alphanumeric
  - National: @ (x’5B’), # (x’7B’), or $ (x’7C’)
  - Period: . (x’4B’)
- The first character must be alphabetic or national.
- Lowercase letters can be used but are internally changed to uppercase.
- The IBM1047 code page is assumed.

6.2.2 The token key data set

The token key data set (TKDS) is a VSAM data set that serves as the repository for cryptographic objects used by z/OS PKCS#11 applications. The TKDS must be created before running applications that intend to create their own tokens and keep persistent object. The ICSF options data set must be updated to point at the TKDS data set.

The following rules govern the creation of the TKDS:

- The token data set must be a key-sequenced VSAM data set with spanned variable-length records.
- The token data set must be allocated on a permanently resident volume.

Important: Private objects, such as keys, are not kept encrypted in the TKDS. Therefore, it is fundamental that strong access control be in place for the data set itself. This can be achieved by defining a proper DATASET profile in RACF and having only the ICSF application be permitted to the TKDS.

Member CSFTKDS in SYS1.SAMPLIB contains a sample JCL for the creation of a TKDS (Figure 6-2).

```
//CSFTKDS   JOB <JOB CARD PARAMETERS>
//*
//DEFINE  EXEC PGM=IDCAMS,REGION=4M
//SYSPRINT DD  SYSOUT=*
//SYSIN DD *
  DEFINE CLUSTER (NAME(CSF.CSFTKDS)        -
                  VOLUMES(XXXXXX)          -
                  RECORDS(100 50)          -
                  RECORDSIZE(2200 32756)   -
                  KEYS(72 0)               -
                  FREESPACE(0 0)           -
                  SPANNED                  -
                  SHAREOPTIONS(2 3))       -
            DATA (NAME(CSF.CSFTKDS.DATA)   -
                  BUFFERSPACE(100000)      -
                  ERASE                    -
                  WRITECHECK)              -
           INDEX (NAME(CSF.CSFTKDS.INDEX))
/*
```

Figure 6-2 JCL example - Creating TKDS in SYS1.SAMPLIB, member CSFTKDS

Note: To optimize performance, ICSF creates and manages a data-space-based cache for objects stored in the TKDS. ICSF also provides sysplex support to maintain consistency of contents between all instances of this cache.

TKDS parameters in the ICSF installation options data set

The ICSF installation option data set contains two parameters related to the token data set:

- TKDSN
- SYSPLEXTKDS

TKDSN identifies the VSAM data set that will be the token data set. The format of this parameter is:

`TKDSN(`*`data_set_name`*`)`

data_set_name is the existing token data set or an empty VSAM data set to be used as the token data set.

Important: If the TKDSN parameter is not specified in the ICSF installation option data set, ICSF provides PKCS#11 services using only the omnipresent token.

SYSPLEXTKDS specifies whether the token data set should have sysplex-wide data consistency. The SYSPLEXTKDS parameter is in effect only if the TKDSN parameter has also been specified. The format of this parameter is:

`SYSPLEXTKDS(YES|NO,FAIL(YES|NO)`

The SYSPLEXTKDS option can be YES or NO:

- If SYSPLEXTKDS(NO,FAIL(fail-option)) is specified, no XCF signalling is performed when an update to a TKDS record occurs.
- If SYSPLEXTKDS(YES,FAIL(fail-option)) is specified, the system is notified of updates made to the TKDS by other members of the sysplex who have also specified SYSPLEXTKDS(YES,FAIL(fail-option)).

The value of FAIL can be YES or NO.

- If FAIL(YES) is specified, ICSF initialization terminates abnormally when a failure creating the TKDS latch set occurs.
- If FAIL(NO) is specified, ICSF initialization processing continues even if the request to create a TKDS latch set fails. The system is not notified of updates to the TKDS by other members of the ICSF sysplex group.

The possible combinations are as follows:

- SYSPLEXTKDS(NO,FAIL(NO))
- SYSPLEXTKDS(YES,FAIL(NO))
- SYSPLEXTKDS(YES,FAIL(YES))

The default value is SYSPLEXTKDS(NO,FAIL(NO).

Serialization of resources for the TKDS

To serialize updates to the TKDS and in-storage copies of TKDS objects, ICSF uses sysplex-wide ENQs on the resource QNAME SYSZTKT and resource RNAME SYSZTCKDS.TKDS*dsn*, where *dsn* is the data set name of the active TKDS. In addition, a latch set is defined for the serialization of accesses to TKDS records.

6.2.3 Controlling access to tokens and objects

The PKCS#11 standard was designed for systems that grant access to token information based on a PIN code. The standard defines two type of users, the standard user (user) and the security officer (SO), each having its own personal identification number (PIN).

z/OS does not use PINs. Instead, profiles in the SAF CRYPTOZ class control access to tokens. For each token, there are two resources in the CRYPTOZ class:

- The resource USER.*token-name* controls the access of the user role to the token.
- The resource SO.*token-name* controls the access of the SO role to the token.

A user's access level to each of these resources (READ, UPDATE, and CONTROL) determines the user's access level to the token and the objects that it stores.

Of the six possible token access levels in z/OS, three are defined by the PKCS#11 standard, and three are unique to z/OS. The PKCS#11 token standard access levels are:

- User R/O: This allows the user to read the token, including its private objects, but the user cannot create a new token or session object or alter existing ones.
- User R/W: This allows the user read/write access to the token objects, including its private objects.
- SO R/W: This allows the user to act as the security officer for the token and to read, create, and alter public objects on the token.

The token access levels unique to z/OS are:

- Weak SO: A security officer who can modify certificate authority certificates contained in a token but not initialize the token (for example, a system administrator who deploys a trust policy for all applications on the system).
- Strong SO: A security officer who can add, generate, or remove private objects in a token but cannot use the keys (for example, an administrator in charge of setting up servers).
- Weak user: A user who has access to everything in a token but cannot alter the trust policy of the token (for example, the user ID assigned to a server daemon).

The user access level to a token is derived from the user's access level to a resource in the SAF CRYPTOZ class (Table 6-1). The CRYPTOZ class resource names are formed from the label of the token being protected: SO.*token-label* for the security officer role and USER.*token-label* for the user role. Generic profiles can be used to protect multiple tokens.

Table 6-1 Token access levels

	SAF access level		
CRYPTOZ resource	**READ**	**UPDATE**	**CONTROL**
SO.*token-label*	Weak SO Can read, create, delete, modify, and use public objects.	SO R/W (1) Same ability as weak SO, plus can create and delete tokens.	Strong SO Same ability as SO R/W, plus can read, create, delete, and modify private objects but not use them (2).
USER.*token-label*	User R/O (1) Can read and use public and private objects (see the note that follows).	Weak user Same ability as user R/O, plus can create, delete, and modify public and private objects. Cannot add, delete, or modify certificate authority objects.	User R/W (1) Same ability as weak user, plus can add, delete, and modify certificate authority objects.

1 Original PKCS#11 roles

2 *Use* is defined as any of the following:

- Performing any cryptographic operation involving the key object (for example, C_Encrypt).
- Searching for key objects using sensitive search attributes.
- Retrieving sensitive key object attributes, such as CKA_VALUE for the secret key.

Here are some general guidelines:

- The CRYPTOZ class supports generic profiles. We suggest that you use this advantage by creating token-naming conventions for the organization that enforces generic profiles (or example, users and applications are required to precede their token name with their user IDs).
- For server applications, grant security officers (in charge of server administration) strong SO access and the server daemons user IDs weak user or user R/W access.
- For applications for which it is not necessary to separate the security officer and user roles, grant user IDs the appropriate access level to both SO and USER profiles.

6.3 z/OS PKCS#11 services

Programs that invoke PKCS#11 functions in z/OS can do so using the C API or by calling the ICSF PKCS#11 callable services.

6.3.1 The z/OS PKCS#11 C API

This API is a subset of the PKCS#11 V2.20 API specification, with the primitive functions needed being backup ICSF services. Dedicated services have been implemented in ICSF to support the PKCS#11 functions and are externalized (as described in “PKCS#11 callable services” on page 130) for applications to access them via CCA calls.

The z/OS C API

This is a subset of the C functions defined in the PKCS#11 V2.20 specification. Most functions are supported, with the exception, for instance, of C_Login and C_Logout, which have been rendered inoperative, as the PKCS#11 implementation in z/OS does not operate with PIN codes.

The supported algorithms and PKCS#11 mechanisms are mapped whenever possible to conventional ICSF services, which are then internally called by the API.

The *ICSF Writing PKCS #11 Applications,* SA23-2231, book in the ICSF documentation provides the following information:

- Functions and mechanisms supported
- Return codes
- Trace and troubleshooting information
- Small test program of PKCS#11 on z/OS

PKCS#11 callable services

ICSF provides a base set of PKCS #11 callable services that users can invoke using the CCA services syntax. These callable services do not require an LE run time. The ICSF PKCS #11 callable services comprise:

- Derive key (CSFPDVK)
- Derive multiple keys (CSFPDMK)
- Generate HMAC (CSFPHMG)
- Generate key pair (CSFPGKP)
- Generate secret key (CSFPGSK)
- Get attribute value (CSFPGAV)
- One-way hash generate (CSFPOWH
- Private key sign (CSFPPKS)
- Pseudo-random function (CSFPPRF)
- Public key verify (CSFPPKV)
- Secret key decrypt (CSFPSKD)
- Secret key encrypt (CSFPSKE)
- Set attribute value (CSFPSAV)
- Token record create (CSFPTRC)
- Token record delete (CSFPTRD)
- Token record list (CSFPTRL)
- Unwrap key (CSFPUWK)
- Verify HMAC (CSFPHMV)
- Wrap key (CSFPWPK)

Calls to the System Authorization Facility (SAF) determine access authorization for the callable services using the CSFSERV class. The user requires READ access to the appropriate CSFSERV class resource. The PKCS#11 services have a SAF resource name of CSF1xxx (for instance, the secret key encrypt service is protected by the CSF1SKE profile in the CSFSERV class).

6.3.2 Supported algorithms

The following PKCS#11 “mechanisms” are supported:

- Encryption
  - CKM_DES_ECB
  - CKM_DES_CBC
  - CKM_DES_CBC_PAD
  - CKM_DES3_ECB
  - CKM_DES3_CBC
  - CKM_DES3_CBC_PAD
  - CKM_RSA_PKCS
  - CKM_RSA_X_509
  - CKM_AES_CBC
  - CKM_AES_ECB
  - CKM_AES_CBC_PAD
  - CKM_AES_GCM[1] (Limited to single part encryption only and for no more than 1048576 bytes of clear text.)
  - CKM_BLOWFISH_CBC
  - CKM_RC4
- Message digest functions
  - CKM_MD2
  - CKM_MD5
  - CKM_SHA_1
  - CKM_SHA224
  - CKM_SHA256
  - CKM_SHA384
  - CKM_SHA512
  - CKM_RIPEMD160
- Signing and message authentication
  - CKM_RSA_X_509
  - CKM_RSA_PKCS
  - CKM_MD5_RSA_PKCS
  - CKM_SHA1_RSA_PKCS
  - CKM_SHA224_RSA_PKCS

[1] Galois Counter Mode

- CKM_SHA256_RSA_PKCS
- CKM_SHA384_RSA_PKCS
- CKM_SHA512_RSA_PKCS
- CKM_DSA
- CKM_DSA_SHA1
- CKM_MD5_HMAC
- CKM_SHA_1_HMAC
- CKM_SHA224_HMAC
- CKM_SHA256_HMAC
- CKM_SHA384_HMAC
- CKM_SHA512_HMAC
- CKM_SSL3_MD5_MAC
- CKM_SSL3_SHA1_MAC
- CKM_MD2_RSA_PKCS
- CKM_ECDSA
- CKM_ECDSA_SHA1

Key management functions

Refer to *Writing PKCS #11 Applications,* SA23-2231, for a description of the PKCS#11 key management functions.

Note: Remember that as of HCR7770:

- Execution of these services resolves “under the covers” into regular CCA calls to the Crypto Express coprocessor, if applicable, and if at least one coprocessor is in operation.
- If no Crypto Express coprocessor is in operation, ICSF will provide the services by software only.

6.3.3 Elliptic Curve Cryptography support

Elliptic Curve Cryptography is an emerging public-key algorithm that will eventually replace RSA cryptography in many applications. ECC is capable of providing digital signature functions an key agreement functions. The new ICSF services provide ECC key generation and key management and provide digital signature generation and verification functions compliant with the ECDSA method described in ANSI X9.62 ("Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)”).

ECC uses keys that are shorter than RSA keys for equivalent strength.

An example of use of Elliptic Curve Cryptography with the ICSF PKCS#11 callable services is given in Chapter 9, “Using Elliptic Curve Cryptography (ECC) on z/OS” on page 245.

6.4 Basics of token administration using ICSF token browser

z/OS PKCS#11 offers different interactive interfaces for token administration:

- The ICSF token browser ISPF panels
- The RACF RACDCERT command or ISPF panels
- The z/OS System SSL gskkyman utility

The differences in these methods are explained in the sections that follow.

The ability to administer the tokens depends on the correct access level defined in the CRYPTOZ SAF class using the resources SO.*token-name* and USER.*token-name*. If the CSFSERV class is active, ICSF also performs access control checks to the callable services that are invoked.

USER. and SO. profiles are not checked to determine access to the omnipresent token. All users have R/W access to the token.

ICSF panels, token browser

To access the token management panels, select option 5, Utility, from the ICSF primary panel to get access to the panel shown in Figure 6-3.

```
------------------------------ ICSF - Utilities ----------------------------
OPTION ===>

Enter the number of the desired option.

  1  ENCODE       -  Encode data
  2  DECODE       -  Decode data
  3  RANDOM       -  Generate a random number
  4  CHECKSUM     -  Generate a checksum and verification and
                     hash pattern
  5  PPKEYS       -  Generate master key values from a pass phrase
  6  PKDSKEYS     -  Manage keys in the PKDS
  7  PKCS11 TOKEN -  Management of PKCS11 tokens









Press ENTER to go to the selected option.
Press END   to exit to the previous menu.
```

Figure 6-3 Utilities panel with token management selection

Select option 7, PKCS11 TOKEN, from the Utilities panel, and the ICSF Token Management main panel displays (Figure 6-4).

```
---------------------- ICSF Token Management - Main Menu -----------------
OPTION ===>

Enter the number of the desired option.

  1  Create a new token
  2  Delete an existing token
  3  Manage an existing token
  4  List existing tokens

  Full or partial token name: _____________________________
```

Figure 6-4 Token Management main panel

The Token Management main panel has options to create, delete, and manage an existing token. All these options require the exact name of the token. Option 4, list existing tokens, can use partial token names (for example, indicating that token name begins with given characters).

If the token name is not given with option 4, all tokens to which the user has appropriate access permission are shown. This includes in all cases the omnipresent token, labelled SYSTOK-SESSION-ONLY (Figure 6-5).

```
--------------------- ICSF Token Management - List Tokens --
 COMMAND ===>

   Select a token to manage(M) or delete(D) then press ENTER

   Press END to return to the previous menu.


 _ SYSTOK-SESSION-ONLY
```

Figure 6-5 List existing tokens without user-created tokens

To create a token, select option 1 and specify the token name. The resource definitions on the CRYPTOZ class might not give you permission to create the token. In our case, we created a token with the name TESTING.TOKEN. The token name must follow the rules given in 6.2.1, “Tokens” on page 126.

There are now two tokens created by ICSF (Figure 6-6).

```
. . . . . .
--------------------- ICSF Token Management - List Tokens ---- Row 1 to 2 of 2
COMMAND ===>                                                  SCROLL ===> PAGE

  Select a token to manage(M) or delete(D) then press ENTER

  Press END to return to the previous menu.

_ SYSTOK-SESSION-ONLY
_ TESTING.TOKEN
****************************** Bottom of data ******************************
```

Figure 6-6 List existing tokens with one user-created token

Users who do not have proper permission to the CRYPTOZ profiles for the TESTING.TOKEN resource will not see the token in the list.

For testing purposes, we deleted the TESTING.TOKEN token by selecting option 2, delete, from the Token Management panel. A pop-up panel displays (Figure 6-7) to confirm that the correct token will be deleted. All certificates, keys, or data, if any, stored in the token container will be lost.

```
---------------------- ICSF - Delete Confirmation --------------------------

Are you sure you want to delete token TESTING.TOKEN?

===> Y_   Enter Y to confirm
```

Figure 6-7 A pop-up panel confirms token delete

A successful message is displayed after the deletion (Figure 6-8). The token is deleted from the in-storage copy and from TKDS. If the SYSPLEXTKDS(YES,FAIL(xxx)) option is in effect in the installation options data set, a sysplex broadcast message is issued, informing other sysplex members of the TKDS update and requesting them to update their in-storage TKDS copy.

```
------------------- ICSF - PKCS11 Token Delete Successful --------------------
COMMAND ===>



Token name ==> TESTING.TOKEN

Token was deleted successfully
```

Figure 6-8 Token delete successful

A new token was created by a test program called PK.TOKEN4. The tokens list now again indicates one user-created token (Figure 6-9).

```
---------------------- ICSF Token Management - List Tokens --
COMMAND ===>

  Select a token to manage(M) or delete(D) then press ENTER

  Press END to return to the previous menu.

_ PK.TOKEN4
_ SYSTOK-SESSION-ONLY
```

Figure 6-9 List available tokens

Selecting the PK.TOKEN4 token with the M option provides a view of the objects contained in the token (Figure 6-10). In this example, the test program generated an RSA key pair. As shown in Figure 6-10 and Figure 6-11 on page 138, attribute data fields related to the keys were left blank by the program. They could have been filled by giving values to these attributes in the test program or can now be edited (some of them) using the token browser.

```
-------------------- ICSF Token Management - Token Details --- Row 1 to 2 of 2
 COMMAND ===>                                                  SCROLL ===> PAGE

 Token name: PK.TOKEN4
 Manufacturer: z/OS PKCS11 API
 Model: HCR7770
 Serial Number: 0
 Number of objects: 2

 Select objects to process then press ENTER

 Press END to return to the previous menu.


--------------------------------------------------------------------------------
 _ Object 1         PUBLIC KEY    PRIVATE: FALSE      MODIFIABLE: TRUE
    LABEL:          <Not-specified>
    SUBJECT:        <Not-specified>
    ID:             <Not-specified>
    RSA MODULUS:    C857334D80F3C7CF106B8A6109DE28750629C110DA3AC70BDB118BFF...
    MODULUS BITS:   512
    USAGE FLAGS:    Enc(T),Verify(T),VerifyR(T),Wrap(T),Derive(T)

 _ Object 2         PRIVATE KEY   PRIVATE: TRUE       MODIFIABLE: TRUE
                                  EXTRACTABLE: TRUE   SENSITIVE: FALSE
    LABEL:          <Not-specified>
    SUBJECT:        <Not-specified>
    ID:             <Not-specified>
    RSA MODULUS:    C857334D80F3C7CF106B8A6109DE28750629C110DA3AC70BDB118BFF...
    USAGE FLAGS:    Dec(T),Sign(T),SignR(T),Unwrap(T),Derive(T)
```

Figure 6-10 Token PK.TOKEN4 contains an RSA key pair

Note the token identification data at the top. They conform to what is indicated in systems where PKCS#11 tokens are real hardware, providing a manufacturer name, a model number, and so on. On z/OS, the Manufacturer field indicates how the token was created and can have one of the following values:

- *ICSF PKCS11 token browser w*hen the token was created using the ICSF token browser.
- *RACF <fmid>* when the token was created using the RACF RACDCERT command.
- *z/OS PKCS11 API* when the token was created using the C API or the gskkyman utility.

The Model field indicates the ICSF FMID, and the Serial number field indicates the version number (if any).

Selecting Object 1 (the public key) yields the detailed information panel shown in Figure 6-11.

```
. . . . . .
 -------------- ICSF Token Management - Public Key Object Details
--------------
 COMMAND ===>                                                SCROLL ===> CSR
 Object 1      from token label: PK.TOKEN4

 Select an Action:
   1  Process select DER fields(*) using external command
         Enter UNIX command pathname (see panel help for details):

   2  Modify one or more fields with the new values specified
   3  Delete the entire object

------------------------------------------------------------------------------
                                                                      More:     +
    OBJECT CLASS:                    PUBLIC KEY
    PRIVATE:                         FALSE
    MODIFIABLE:                      TRUE
    LABEL:                           <not-specified>
                         New value:
    TRUSTED:                         TRUE
    SUBJECT*:                        <not-specified>

    ID:                              <not-specified>
                         New value:
    KEY TYPE:                        RSA
    START DATE:                      <not-specified>
                         New value:               YYYYMMDD
    END DATE:                        <not-specified>
                         New value:               YYYYMMDD
    DERIVE:                          TRUE
 LOCAL:                          TRUE
 KEY GEN MECHANISM:              UNAVAILABLE INFORMATION
 ENCRYPT:                        TRUE
                      New value: FALSE
 VERIFY:                         TRUE
                      New value: FALSE
 VERIFY RECOVER:                 TRUE
                      New value: FALSE
 WRAP:                           TRUE
                      New value: FALSE
 FIPS140:                        FALSE
 APPLICATION:                    <not-specified>
 MODULUS BITS:                   512
 PUBLIC EXPONENT:
    010001
```

Figure 6-11 Public key object's details

6.5 Exploiting the PKCS#11 services

In this section we share our experience with setting up and exploiting, to a limited extent, the z/OS PKCS#11 services, focusing on illustrating what the z/OS PKCS#11 concepts implementation translates into for z/OS security administrators and applications programmers.

6.5.1 Making the services available

Access to the PKCS#11 services requires that ICSF has been successfully started, with or without a TKDS defined. If a TKDS is not defined, the services will apply to non persistent objects.

However, it is also necessary to have the CRYPTOZ class of resources activated in RACF. If not active, the following will be displayed in the ICSF startup messages:

```
CSFM012I NO ACCESS CONTROL AVAILABLE FOR CRYPTOZ RESOURCES. ICSF PKCS11 SERVICES
DISABLED.
```

The CRYPTOZ class must be activated, and its resources RACLISTed with the following RACF administration commands, and ICSF restarted:

```
SETROPTS CLASSACT(CRYPTOZ)
SETROPTS RACLIST(CRYPTOZ)
```

6.5.2 Learning with the tespkcs11 program

The testpkcs11 program is, as the name implies, a test program provided in z/OS to quickly check out the PKCS#11 services. It is a C program provided in the `/usr/lpp/pkcs11` directory. Refer to *Writing PKCS #11 Applications* for information about how to prepare and use the programs. What the program does is duly indicated by the messages that it displays in the z/OS UNIX shell (Figure 6-12).

```
KAPPELE @ SC64:/Z1BRA1/usr/lpp/pkcs11/bin>./testpkcs11 -t pk.token4
Getting the PKCS11 function list...
Initializing the PKCS11 environment...
Creating the temporary token...
Opening a session...
Generating keys. This may take a while...
Enciphering data...
Deciphering data...
Destroying keys...
Closing the session...
Deleting the temporary token...
Test completed successfully!
```

Figure 6-12 Execution of the testpkcs11 program

The program is invoked by issuing a **`testpkcs11 -t <token_name>`** command. An RSA key pair is generated and data are encrypted with the public key, and then decrypted with the private key and the result is compared with the initial data.

It appears that, even with very limited knowledge of the C language, the program can be easily modified for the purpose of exploring specific situations that help provide a better understanding of z/OS PKCS#11 principles of operation.

Note, however, that the program requires ICSF to be active, but also a TKDS to be defined and proper setup of RACF profiles in the CRYPTOZ class of resources. Failing to meet these conditions results in the error conditions that we discuss in the following sections.

Lesson 1: ICSF should be active and a TKDS defined

If these conditions are not fulfilled, the program fails with the message shown inFigure 6-13.

```
KAPPELE @ SC64:/Z1BRA1/usr/lpp/pkcs11/bin>./testpkcs11 -t pk.token4
Getting the PKCS11 function list...
Initializing the PKCS11 environment...
   C_Initialize returned:  224 (0xe0) CKR_TOKEN_NOT_PRESENT - ICSF is not
active or not configured for TKDS operations
```

Figure 6-13 testpkcs11 test program and no TKDS defined

Lesson 2: Properly defining profiles in the CRYPTOZ class

In this section we recount our train-and-error approach to correctly define the RACF profiles.

Warning: We found that the test program might fail to properly delete the created token when terminated by a failure condition. Trying to re-run the program gives the messages shown in Figure 6-14. The ICSF token browser can then be used to delete the token before attempting a new run.

```
KAPPELE @ SC64:/Z1BRA1/usr/lpp/pkcs11/bin>./testpkcs11 -t pk.token2
Getting the PKCS11 function list...
Initializing the PKCS11 environment...
Creating the temporary token...
   C_InitToken #1 returned:  7 (0x7) CKR_ARGUMENTS_BAD
   Make sure your the token name you specified meets ICSF rules:
   Contains only alphanumeric characters, nationals (@#$), and periods
   The first character cannot be a numeric or a period
```

Figure 6-14 testpkcs11 program - Token already existing problem

No CRYPTOZ profile defined

This was our initial status when trying testpkcs11. This yielded the message shown in Figure 6-15.

```
Getting the PKCS11 function list...
Initializing the PKCS11 environment...
Creating the temporary token...
   C_InitToken #1 returned:  225 (0xe1) CKR_TOKEN_NOT_RECOGNIZED - You are not
authorized to perform the token operation
```

Figure 6-15 No profile defined in the CRYPTOZ class

Defining a security officer profile

We then fist defined a security officer (SO) for our arbitrary test token (PK.TOKEN4, not existing yet), as follows:

```
RDEFINE CRYPTOZ SO.PK.TOKEN4 UACC(NONE)
PERMIT SO.PK.TOKEN4 CLASS(CRYPTOZ) ID(KAPPELE) ACCESS(UPDATE)
SETROPTS RACLIST(CRYPTOZ) REFRESH
```

Running the test again still resulted in an error (Figure 6-16), and the following RACF message issued in the system's syslog:

```
ICH408I USER(KAPPELE ) GROUP(SYS1    ) NAME(PATRICK KAPPELER    )
SO.PK.TOKEN1 CL(CRYPTOZ )
INSUFFICIENT ACCESS AUTHORITY ACCESS INTENT(CONTROL)  ACCESS ALLOWED(UPDATE
```

Table 6-1 on page 129 provides the explanation: A security officer with an UPDATE access level to a SO.<token_name> profile can perform the following actions against the token:

- Can read, create, delete, modify, and use public objects.
- Can create and delete tokens.

If we had given the CONTROL access level to the security officer, he could have also been able to read, create, delete, and modify private objects (but without being able to use them).

This is the privilege that we were missing, because the program running under the identity permitted to the SO.<*token_name*> profile was to generate a key pair in the created token.

```
KAPPELE @ SC64:/Z1BRA1/usr/lpp/pkcs11/bin>./testpkcs11 -t pk.token4
Getting the PKCS11 function list...
Initializing the PKCS11 environment...
Creating the temporary token...
Opening a session...
Generating keys. This may take a while...
   C_GenerateKeyPair #1 returned:  225 (Oxe1) CKR_TOKEN_NOT_RECOGNIZED - You
are not authorized to perform the token operation
Deleting the temporary token...
   C_InitToken #2 (for delete) returned:  182 (Oxb6) CKR_SESSION_EXISTS
KAPPELE @ SC64:/Z1BRA1/usr/lpp/pkcs11/bin>
```

Figure 6-16 testpkcs11 program - The security officer does not have proper authority

We then gave the security officer identity the CONTROL access level to the PK.TOKEN4 resource:

```
PERMIT SO.PK.TOKEN4 CLASS(CRYPTOZ) ID(KAPPELE) ACCESS(CONTROL)
SETROPTS RACLIST(CRYPTOZ) REFRESH
```

However, we still could not run the program properly (Figure 6-17).

```
Getting the PKCS11 function list...
Initializing the PKCS11 environment...
Creating the temporary token...
Opening a session...
Generating keys. This may take a while...
Enciphering data...
Deciphering data...
   C_Decrypt #1 returned:  225 (0xe1) CKR_TOKEN_NOT_RECOGNIZED - You are not
authorized to perform the token operation
Deleting the temporary token...
```

Figure 6-17 testpkcs11 program - Not authorized to use keys

The messages in Figure 6-17 tell us that the data string was successfully encrypted, but the decryption operation failed because of a missing authorization. The explanation is that the encryption step is performed using the RSA public key, a public object that a PKCS#11 security officer can use, however, the decryption uses the RSA private key, a private object that a security officer, whatever his access level is, cannot use.

We are therefore at a point where we are required to give the identity running the program a PKCS#11 user privilege.

Defining a USER profile

Looking again at Table 6-1 on page 129, we decided to define a USER.*<token_name>* profile in the CRYPTOZ class, using the following RACF commands:

```
RDEFINE CRYPTOZ USER.PK.TOKEN4 UACC(NONE)
PERMIT USER.PK.TOKEN4 CLASS(CRYPTOZ) ID(KAPPELE) ACCESS(UPDATE)
SETROPTS RACLIST(CRYPTOZ) REFRESH
```

This enabled us to now run the program successfully (Figure 6-18).

```
KAPPELE @ SC64:/Z1BRA1/usr/lpp/pkcs11/bin>./testpkcs11 -t pk.token4
Getting the PKCS11 function list...
Initializing the PKCS11 environment...
Creating the temporary token...
Opening a session...
Generating keys. This may take a while...
Enciphering data...
Deciphering data...
Destroying keys...
Closing the session...
Deleting the temporary token...
Test completed successfully!
```

Figure 6-18 estpkcs11 - Meeting all the conditions

Lesson 3: Making objects persistent

As clearly stated in the tespkcs11 program messages, the objects and the token are destroyed at the end of the program execution. People curious about seeing how these

objects look do not have any z/OS built-in tools to visualize the temporary contents of the token.

Hence, our intent to modify the tespkcs11 program so that the token and the objects that it contains remain persistent in the TKDS after executing the program. The ICSF token browser could then be used to look at these objects.

Removing the key destroy and token delete steps

This appeared to be an easy modification, even for non-C-literate. The involved program steps were just commented out, leaving as execution steps the ones shown in Figure 6-19.

```
KAPPELE @ SC64:/u/kappele/pkcs11>modpkcs11 -t pk.token4
Getting the PKCS11 function list...
Initializing the PKCS11 environment...
Creating the temporary token...
Opening a session...
Generating keys. This may take a while...
Enciphering data...
Deciphering data...
Closing the session...
Test completed successfully!
```

Figure 6-19 ttestpkcs11 program - Key destroy and token delete removed

However, looking at the token content token after successful program execution showed no objects stored in the token (Figure 6-20).

```
-------------------- ICSF Token Management - Token Details -------------------
COMMAND ===>                                                  SCROLL ===> PAGE

Token name: PK.TOKEN4
Manufacturer: z/OS PKCS11 API
Model: HCR777O
Serial Number: O
Number of objects: 0
```

Figure 6-20 testpkcs11 modified program - Still no objects left in the token

Making the key objects persistent

Looking into the *Writing PKCS #11 Applications* ICSF book, there are tables specifying what attributes can be given to objects used in z/OS PKCS#11 applications. There are specific tables for attributes that can be given to data objects, X.509 certificates objects, secret key objects, and so on.

The tables for the public key object and the private key object attributes indicate a CKA_TOKEN attribute, which is specified as a boolean type of data, with:

```
Default value on create is FALSE. Object hardened to the TKDS if TRUE. An
application can specify the value when the object is created (or generated) only.
```

The testpkcs11 program was therefore modified to add the CKA_TOKEN attribute to both the public and private key objects. The public and private key sets of attributes (the keys' "templates") have been modified in the tespkcs11 program source code (Figure 6-21).

```
/* Attributes for the public key to be generated */
CK_ATTRIBUTE pub_tmpl[] = {
      {CKA_MODULUS_BITS, &bits, sizeof(bits) },
      {CKA_ENCRYPT, &true, sizeof(true) },
      {CKA_VERIFY, &true, sizeof(true) },
      {CKA_TOKEN, &true, sizeof(true) },
      {CKA_PUBLIC_EXPONENT, &pub_exp, sizeof(pub_exp) }
};
/* Attributes for the private key to be generated */
CK_ATTRIBUTE priv_tmpl[] =
{
      {CKA_DECRYPT, &true, sizeof(true) },
      {CKA_TOKEN, &true, sizeof(true) },
      {CKA_SIGN, &true, sizeof(true) }
};
```

Figure 6-21 testpkcs11 program - Making the key objects persistent (1 of 2)

The C_GenerateKeyPair function parameters also had to be changed to accommodate for the additional attributes. The public template attributes count was modified from 4 to 5, and the private key template attribute count from 2 to 3. Figure 6-22 shows the resulting statement.

```
rc = funcs->C_GenerateKeyPair( session, &mech,
                               pub_tmpl, 5,
                               priv_tmpl, 3,
                               &publ_key, &priv_key );
```

Figure 6-22 testpkcs11 program - Making the key objects persistent (2 of 2)

Re-running the testpkcs11 program with these modifications met the objective of keeping the key objects persistent in the token. This was verified using the ICSF Token Browser (Figure 6-23).

```
-------------------- ICSF Token Management - Token Details --- Row 1 to 2 of 2
 COMMAND ===>                                                  SCROLL ===> PAGE

 Token name: PK.TOKEN4
 Manufacturer: z/OS PKCS11 API
 Model: HCR777O
 Serial Number: O
 Number of objects: 2

 Select objects to process then press ENTER

 Press END to return to the previous menu.


--------------------------------------------------------------------------------
 _ Object 1         PUBLIC KEY    PRIVATE: FALSE      MODIFIABLE: TRUE
    LABEL:          <Not-specified>
    SUBJECT:        <Not-specified>
    ID:             <Not-specified>
    RSA MODULUS:    C857334D8OF3C7CF1O6B8A6109DE28750629C11ODA3AC7OBDB118BFF...
    MODULUS BITS:   512
    USAGE FLAGS:    Enc(T),Verify(T),VerifyR(T),Wrap(T),Derive(T)

 _ Object 2         PRIVATE KEY   PRIVATE: TRUE       MODIFIABLE: TRUE
                                  EXTRACTABLE: TRUE   SENSITIVE: FALSE
    LABEL:          <Not-specified>
    SUBJECT:        <Not-specified>
    ID:             <Not-specified>
    RSA MODULUS:    C857334D8OF3C7CF1O6B8A6109DE28750629C11ODA3AC7OBDB118BFF...
    USAGE FLAGS:    Dec(T),Sign(T),SignR(T),Unwrap(T),Derive(T)
```

Figure 6-23 testpkcs11 program - Looking at the now persistent token contents

Selecting an object allows you to see the object's details, as shown in Figure 6-24 for the private key object.

```
-------------- ICSF Token Management - Private Key Object Details -------------
COMMAND ===>                                                  SCROLL ===> CSR
Object 2      from token label: PK.TOKEN4

Select an Action:
  1  Process select DER fields(*) using external command
       Enter UNIX command pathname (see panel help for details):

  2  Modify one or more fields with the new values specified
  3  Delete the entire object
--------------------------------------------------------------------------------
                                                                        More:     +
   OBJECT CLASS:                 PRIVATE KEY
   PRIVATE:                      TRUE
   MODIFIABLE:                   TRUE
   LABEL:                        <not-specified>
                      New value:
   SUBJECT*:                     <not-specified>

   ID:                           <not-specified>
                      New value:
   KEY TYPE:                     RSA
   START DATE:                   <not-specified>
                      New value:               YYYYMMDD
   END DATE:                     <not-specified>
                      New value:               YYYYMMDD
   DERIVE:                       TRUE
   LOCAL:                        TRUE
LOCAL:                        TRUE
KEY GEN MECHANISM:            UNAVAILABLE INFORMATION
DECRYPT:                      TRUE
                    New value: FALSE
SIGN:                         TRUE
                    New value: FALSE
SIGN RECOVER:                 TRUE
                    New value: FALSE
UNWRAP:                       TRUE
                    New value: FALSE
EXTRACTABLE:                  TRUE          (Cannot be changed from FALSE
                    New value: FALSE          to TRUE)
SENSITIVE:                    FALSE         (Cannot be changed from TRUE
                    New value: TRUE           to FALSE)
ALWAYS SENSITIVE:             FALSE
NEVER EXTRACTABLE:            FALSE
FIPS140:                      FALSE
APPLICATION:                  <not-specified>
PRIVATE EXPONENT:             Not displayable
PRIME 1:                      Not displayable
PRIME 2:                      Not displayable
EXPONENT 1:                   Not displayable
EXPONENT 2:                   Not displayable
COEFFICIENT:                  Not displayable
PUBLIC EXPONENT:
   010001
MODULUS:
   C857334D80F3C7CF106B8A6109DE28750629C110DA3AC70BDB118BFFC43BE566
   BF3B3E63EAA242969F29BE3D8BBD4DF4BC68EF029412D62855CCDF4BFDFD25DB
```

Figure 6-24 testpkcs11 program - Looking at the private key persistent object

Lesson 5: What ICSF services are called under the covers

We ran the testpkcs11 program without our modifications and processed the ICSF component trace with the COUNTS options in IPCS to get a list of the ICSF services invoked during this run. This yielded the IPCS output shown in Figure 6-25.

```
COMPONENT TRACE FULL FORMAT
 COMP(CSF)
 OPTIONS((COUNTS))
ICSF COUNTS FROM CTRACE:
SERVICE CALLS_FOUND = 00000009
FAILING SERVICES    = 00000000
SERVICE   #SUCCESS   #FAILED
CSFVCTRL  00000001   00000000
CSFVCIQA  00000001   00000000
CSFVCTRC  00000001   00000000
CSFNCGKP  00000001   00000000
CSFNCPKV  00000001   00000000
CSFNCPKS  00000001   00000000
CSFVCTRD  00000003   00000000
```

Figure 6-25 ICSF services called by the C Language Environment®

This is to be compared with the list of externalized PKCS#11 callable services that we discuss in the next section, using the last three letters of the service name (for instance, CSFNCGKP in the trace refers to the CSFPGKP (Generate key pair) callable service. Note that CSFVCIQA refers to the ICSF Query Algorithm (CSFIQA) service, which is not peculiar to the PKCS#11 support.

This closes our first set of trials with the testpkcs11 program. We again used the program to explore the z/OS and ICSF FIPS 140-2 support, as explained in 6.6, “Operating in compliance with FIPS 140-2” on page 153.

6.5.3 Using the ICSF PKCS#11 callable services

The initial implementation of z/OS PKCS#11 with ICSF HCR7740 and z/OS V1R9 was externalizing a few ICSF callable services that were dedicated to TKDS token management functions. Other services that the C Language Environment was calling were not available to ICSF users. At ICSF HCR7770, the set of externalized PKCS#11 callable services has been expanded so that non-LE programs can invoke PKCS#11 functions such as key generation, encryption, and digital signature. The callable services listed below are available at ICSF HCR7770:

- Derive multiple keys (CSFPDMK)
- Derive key (CSFPDVK)
- Get attribute value (CSFPGAV)
- Generate key pair (CSFPGKP)
- Generate secret key (CSFPGSK)
- Generate HMAC (CSFPHMG)
- Verify HMAC (CSFPHMV)
- One-way hash generate (CSFPOWH)
- Private key sign (CSFPPKS)
- Public key verify (CSFPPKV)
- Pseudo-random function (CSFPPRF)
- Set attribute value (CSFPSAV)

- Secret key decrypt (CSFPSKD)
- Secret key encrypt (CSFPSKE)
- Token record create (CSFPTRC)
- Token record delete (CSFPTRD)
- Token record list (CSFPTRL)
- Unwrap key (CSFPUWK)
- Wrap key (CSFPWPK)ICSF Query Algorithm (CSFIQA)

These services are described in the *ICSF Application Programmer's Guide*, SA22-7522.

Note: As previously mentioned, users will find out that in many cases a thorough understanding of the PKCS #11 V2.20: CRYPTOGRAPHIC TOKEN INTERFACE STANDARD is required. The standard can be downloaded from:

http://www.rsasecurity.com/rsalabs/

Syntax of the service calls

Taking as an example the CSFPGKP (Generate Key Pair) service, the format of the call is defined as shown in Figure 6-26 in the *ICSF Application Programmer's Guide*, SA22-7522.

```
CALL CSFPGKP(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             token_handle,
             rule_array_count,
             rule_array,
             public_key_attribute_list_length,
             public_key_attribute_list,
             public_key_object_handle,
             private_key_attribute_list_length,
             private_key_attribute_list,
             private_key_object_handle)
```

Figure 6-26 The CSFPGKP (Generate Key Pair) call format

The usual call parameters are specified, namely:

- Return code output field pointer
- Reason code output field pointer
- Exit data length input field pointer
- Exit data input field pointer
- Rule array count input field pointer
- Rule array input field pointer

Then specific PKCS#11 services call parameters have to be explained.

The token and object handle

Whenever a new PKCS#11 object is created it is given by ICSF, as per the PKCS #11 CRYPTOGRAPHIC TOKEN INTERFACE STANDARD, an identifier called a *handle*. Further reference to the object in a program is achieved using this object's handle. For example, a program first can create a token and use the ICSF-generated handle to refer to the token when it comes to generate keys in this token. Note that the generated keys will also be given

their own handle, so that they can be referred to as needed for encryption or decryption operations.

In ICSF, the object handle is a 44-byte identifier with the layout shown in Table 6-2.

Table 6-2 Layout

Name of token or object	Sequence number	ID
32 bytes	8 bytes	4 bytes

The token name in the first 32 bytes of the handle is provided by the PKCS #11 application when the token or object is created. The first character of the name must be alphabetic or a national character (#, $, or @). Each of the remaining characters can be alphanumeric, a national character (#, $, or @), or a period (.)

The sequence number is a hexadecimal number stored as the EBCDIC representation of eight hexadecimal numbers. The sequence number field in a token is EBCDIC blanks. The token record contains a last-used sequence number field, which is incremented each time that an object associated with the token is created. This sequence number value is placed in the handle of the newly created object.

The ID field is four characters. The first character contains an EBCDIC “T” if the handle belongs to a token object, “S” if the handle belongs to a session object, or blank if the handle belongs to a token. The last three characters must be EBCDIC blanks.

Specifying the object’s attributes

PKCS#11 objects are given attributes when they are created that specify the nature and properties of the object. As an example, the attributes given to the RSA keys generated when running the testpkcs11 programs are shown in Figure 6-21 on page 144. The value given to the private key attributes CKA_DECRYPT or CKA_SIGN specify whether the key can be used to decrypt data or to sign data.

The *Writing PKCS#11 Applications* ICSF book specifies the objects and attributes that ICSF supports. Tables indicate:

- The name of the attribute (for instance, CKA_TOKEN)
- The data type (in the case of CKA_TOKEN: CK_BBOOL (boolean value “true” or “false”))
- Relevant usage notes, such as “Default value on create is FALSE. Object hardened to the TKDS if TRUE. An application can specify the value when the object is created (or generated) only”

Attributes to be provided when calling ICSF services are in the format shown in Table 6-3.

Table 6-3 Format

Attribute name	Length of value(n)	Value
4 bytes	2 bytes	n bytes

Attention: The literal attribute name indicated in the ICSF books is not to fit within the 4-byte field in the attribute structure that ICSF expects. Instead, a fixed hexadecimal value is specified that indicates what the attribute name is. The hexadecimal values to be used can be found in two places:

- In the PKCS #11 v2.20 Standard. Look for the Manifest Constants pages.
- In the C header file provided with z/OS at `/usr/lpp/pkcs11/csnpdefs.h`.

As an example of the hexadecimal values to use for attribute names and their value, looking into the standard, or into the `csnpdefs.h header` file, yields the following:

- The CKA_MODULUS_BITS attribute is to be represented by the 0x00000121 hexadecimal value. According to the format indicated above, the structure that will specify a 1024-bit modulus length will be:

  ```
  0x00000121
  0x0004
  0x00000400
  ```

  In System z Assembler Language this translates into:

  ```
  DC X'00000121'    THE CONSTANT VALUE FOR THE ATTRIBUTE NAME
  DC X'0004'        THE LENGTH OF THE ATTRIBUTE VALUE
  DC X'00000400'    THE VALUE IS 1024 DECIMAL
  ```

- Likewise, the CKA_ENCRYPT attribute will be coded as follows, in System z Assembler Language:

  ```
  DC X'00000104'    THE CONSTANT VALUE FOR THE ATTRIBUTE NAME
  DC X'0001'        THE LENGTH OF THE ATTRIBUTE VALUE
  DC X'01'          THE BOOLEAN VALUE IS "TRUE" ("0" WOULD BE "FALSE")
  ```

Specifying the attribute list length

ICSF also requires the list of provided attributes to be specified in terms of number of attributes and length of the list.

Coding the service call

Here is a concrete example of what should be specified when calling the ICSF CSFPGKP service. We are assuming here that a user wants to generate an RSA key pair that is to be stored in an already created token.

See the following documents for more information:

- The *ICSF Application Programmer's Guide* for the syntax of the service call and other pertinent information
- The *ICSF Writing PKCS#11 Applications* for information about mechanisms and attributes supported by ICSF
- The PKCS #11 v2.20 Standard for the constant values to be used to specify mechanisms and attributes

Token handle

As we mentioned above, the assumption is that the PKCS#11 token that we want to use has already been created (using the ICSF CSFPTRC service, for example). Therefore, a 44-byte token handle has already been created and is referred to in the CSFPGKP call.

Rule array

There is no rule array keyword to be specified for the CSFPGKP service. The rule_array_count should therefore be specified as 0.

Public key attributes

The attributes that we want to give to the public key are:

- CKA_CLASS with a value of “CKO_PUBLIC_KEY”
- CKA_KEY_TYPE with a value of “CKK_RSA”
- CKA_MODULUS_BITS with a value of “1024”

  This is the length of the key.
- CKA_ENCRYPT with a value of “TRUE”

  The key can be used to encrypt data.
- CKA_VERIFY with a value of “TRUE”

  The key can be used to verify a digital signature.
- CKA_PUBLIC_EXPONENT with a value of “3”

  We arbitrarily fix the value of the public exponent. We could have left the PKCS#11 service to generate a random public exponent.
- CKA_TOKEN with a value of “TRUE”

  To request the key object to be stored in the token.

The public key attribute list

Taking as an example a program coded in System z Assembler Language, the attribute list and its length would be:

```
PUB_ATTL DC    AL4(PUBLEND-PUBLSTRT)   ATTRIBUTE LIST LENGTH
PUBLSTRT EQU   *
PUB_ATT  DC    X'0007'           NUMBER OF ATTRIBUTES
*
         DC    X'00000000'       CKA_CLASS
         DC    X'0004'          LENGTH OF VALUE
         DC    X'00000002'      CKO_PUBLIC_KEY
*
         DC    X'00000100'       CKA_KEY_TYPE
         DC    X'0004'          LENGTH OF VALUE
         DC    X'00000000'      CKK_RSA
*

         DC    X'00000121'       CKA_MODULUS_BITS
         DC    X'0004'          LENGTH OF VALUE
         DC    X'00000400'      1024 BITS
*
         DC    X'00000104'       CKA_ENCRYPT
         DC    X'0001'          LENGTH OF VALUE
         DC    X'01’            TRUE
*
         DC    X'0000010A'       CKA_VERIFY
         DC    X'0001'          LENGTH OF VALUE
         DC    X'01’            TRUE
*
         DC    X'00000122'       CKA_PUBLIC_EXPONENT
```

```
         DC    X'0004'          LENGTH OF VALUE
         DC    X'00000003'            3
*
         DC    X'00000001'       CKA_TOKEN
         DC    X'0001'          LENGTH OF VALUE
         DC    X'01'            TRUE
PUBLEND  EQU   *
```

Private key attributes

The attributes that we want to give to the private key are:

- CKA_CLASS with a value of “CKO_PRIVATE_KEY”
- CKA_KEY_TYPE with a value of “CKK_RSA”
- CKA_DECRYPT with a value of “TRUE”

  The key can be used to decrypt encrypted data.
- CKA_SIGN with a value of “TRUE”

  The key can be used to sign data.
- CKA_TOKEN with a value of “TRUE”

  This is to request the key object to be stored in the token.

The private key attribute list

Taking as an example a program coded in System z Assembler Language, the attribute list and its length would be:

```
PRV_ATTL DC    AL4(PRVLEND-PRVLSTRT)   ATTRIBUTE LIST LENGTH
PRVLSTRT EQU   *
PRV_ATT  DC    X'0005'           NUMBER OF ATTRIBUTES
*
         DC    X'00000000'       CKA_CLASS
         DC    X'0004'          LENGTH OF VALUE
         DC    X'00000003'      CKO_PRIVATE_KEY
*
         DC    X'00000100'       CKA_KEY_TYPE
         DC    X'0004'          LENGTH OF VALUE
         DC    X'00000000'      CKK_RSA
*

         DC    X'00000105'       CKA_DECRYPT
         DC    X'0001'          LENGTH OF VALUE
         DC    X'01'            TRUE
*
         DC    X'00000108'       CKA_SIGN
         DC    X'0001'          LENGTH OF VALUE
         DC    X'01'            TRUE
*
         DC    X'00000001'       CKA_TOKEN
         DC    X'0001'          LENGTH OF VALUE
         DC    X'01'            TRUE
PRVLEND  EQU   *
```

Key handles

The key handles are fields that will be filled by ICSF with a unique value when generating the keys. For using the generated keys, the program should then point at its handle.

An example of a definition of these fields, again in System z Assembler Language, would be:

```
PUB_HDL  DC    CL44' '           PUBLIC KEY HANDLE
PRV_HDL  DC    CL44' '           PRIVATE KEY HANDLE
```

After executing the service call

The generated key pair can now be looked at using the ICSF token browser, displaying in our case the information shown in Figure 6-27.

```
-------------------- ICSF Token Management - Token Details --- Row 1 to 2 of 2
COMMAND ===>                                                  SCROLL ===> PAGE

Token name: PK.TOKEN4
Manufacturer: MANUFACTURED BY P.KAPPELER
Model: MODEL 0000
Serial Number: S/N 000000
Number of objects: 2

Select objects to process then press ENTER

Press END to return to the previous menu.

--------------------------------------------------------------------------------
_ Object 1         PUBLIC KEY    PRIVATE: FALSE      MODIFIABLE: TRUE
   LABEL:          <Not-specified>
   SUBJECT:        <Not-specified>
   ID:             <Not-specified>
   RSA MODULUS:    E1FC68182E37E430223A861940AA28F08FA5646AC8851B6BA000C809...
   MODULUS BITS:   1024
   USAGE FLAGS:    Enc(T),Verify(T),VerifyR(T),Wrap(T),Derive(T)

_ Object 2         PRIVATE KEY   PRIVATE: TRUE       MODIFIABLE: TRUE
                                 EXTRACTABLE: TRUE   SENSITIVE: FALSE
   LABEL:          <Not-specified>
   SUBJECT:        <Not-specified>
   ID:             <Not-specified>
   RSA MODULUS:    E1FC68182E37E430223A861940AA28F08FA5646AC8851B6BA000C809...
   USAGE FLAGS:    Dec(T),Sign(T),SignR(T),Unwrap(T),Derive(T)
```

Figure 6-27 Token contents after executing the ICSF CSFPGKP service

6.6 Operating in compliance with FIPS 140-2

The National Institute of Standards and Technology (NIST) is the U.S. federal technology agency that works with industry to develop and apply technology, measurements, and standards. One of the standards published by NIST is the Federal Information Processing Standard Security Requirements for Cryptographic Modules, referred to as FIPS 140-2.

FIPS 140-2 provides a standard that can be required by organizations that specify that cryptographic-based security systems are to be used to provide protection for sensitive or valuable data. The standard covers what algorithms are deemed to provide appropriate protection and what minimum key lengths should be used.

z/OS implementation of PKCS #11 cryptography is designed to meet FIPS 140-2 Level 1 criteria and can optionally be configured to operate in compliance with the specifications in the standard. Applications that need to comply with the FIPS 140-2 standard can therefore use the z/OS PKCS #11 services in a way that allows only the cryptographic algorithms (including key sizes) approved by the standard and denies access to the algorithms that are not FIPS-approved.

6.6.1 Preparing to comply with FIPS 140-2

There are two points to consider when preparing to use z/OS PKCS#11 in compliance with FIPS 140-2:

- ICSF must be set up to use the specific load modules that are designed to be FIPS 140-2 compliant. This is accomplished by specifying the FIPSMODE option in the options data set. This option allows you to specify that all applications will have to call the PKCS#11 services in a way that complies with the standard, or the service will be denied. Or that only applications that execute in specific conditions will have to comply with the standard. Other applications can then diverge from the standard. This is explained below.
- The ICSF FIPS 140-2-compliant load modules are digitally signed by IBM. Installations can check the integrity of these modules at program load time by using the optional digital signature verification function that RACF offers beginning with z/OS V1R11. Note that not running this verification will not impact the execution of the ICSF FIPS 140-2-compliant PKCS#11 services. However, the installation might not in that case rely on a guaranteed integrity of the code that ICSF loads and executes

6.6.2 Verification of the ICSF load module digital signature

The load module that contains the FIPS 140-2-compliant cryptographic functions is SYS1.SIEALNKE(CSFINPV2), and this load module is digitally signed when it is shipped from IBM.

The explanation of the setup steps to be performed is spread through the *ICSF Writing PKCS#11 Applications*, the *z/OS Security Server RACF Security Administrator's Guide*, and the *z/OS Security Server RACF System Programmer's Guide*. A summary of these steps follows.

Preparing RACF to perform signature verification

RACF should have been set up to perform signature verification. This is achieved by:

1. Creating a key ring that contains the certificate (and therefore the public key) used by RACF to verify the program's signature.

   In the case of IBM-signed programs, the certificate is already available in the RACF database under the label STG Code Signing CA.

   These are RACF commands to issue for creating the key ring, which we call here "FIPS142.SIGNVER.RING", and connecting the certificate to the ring after ensuring that its status is set to TRUST:

```
RACDCERT ID(KAPPELE) ADDRING(FIPS142.SIGNVER.RING)
RACDCERT CERTAUTH ALTER(LABEL('STG Code Signing CA')) TRUST
RACDCERT ID(KAPPELE) CONNECT(CERTAUTH LABEL('STG Code Signing CA')
```

2. Indicating in the FACILITY profile below what key ring should be looked at for the verification key. This is achieved with the following RACF commands:

```
RDEFINE FACILITY IRR.PROGRAM.SIGNATURE.VERIFICATION
APPLDATA('KAPPELE/FIPS1402.SIGNVER.RING'
SETROPTS RACLIST(FACILITY) REFRESH
```

Attention: Specify only uppercase characters in the key ring name, as the name will be later entered in the APPLDATA field of a FACILITY profile.

3. Preparing to verify the signature of program IRRPVERS that RACF uses to verify signatures of programs. This is done by defining a PROGRAM profile for IRRPVERS as follows:

```
RDEFINE PROGRAM IRRPVERS ADDMEM('SYS1.SIEALNKE'//NOPADCHK) UACC(READ)
SIGVER(SIGREQUIRED(YES) FAILLOAD(ANYBAD) SIGAUDIT(ANYBAD))
SETROPS WHEN(PROGRAM) REFRESH
```

4. Loading the IRRPVERS program for the first time and verifying its signature. This is done by the RACF system programmer running the IRRVERLD program. See the *RACF System Programmer's Guide*. When this step is completed, other programs can be verified provided that they are properly defined in a profile in the PROGRAM class.

Instructing RACF to perform signature verification on CSFINPV2

To have RACF verify the program's signature at load time, a PROGRAM profile has to be defined for CSFINPV2 itself, as follows:

```
RDEFINE PROGRAM CSFINPV2 ADDMEM('SYS1.SIEALNKE'//NOPADCHK) UACC(READ)
SIGVER(SIGREQUIRED(YES) FAILLOAD(ANYBAD) SIGAUDIT(ANYBAD))
SETROPTS WHEN(PROGRAM) REFRESH
```

When ICFS is now to load the module CSFINPV2 at startup, the module signature will be checked by RACF. In case of unsuccessful signature verification, RACF issues the following message:

```
ICH440I Program signature error <return code/reason code> for program CSFINPV2 in
library SYS1.SIEALNKE. The program was not loaded
```

Note: Not performing signature verification will not preclude executing the FIPS-compliant services in ICSF. However, the integrity of compliance could not be proven in this case.

6.6.3 Directing ICSF to provide FIPS 140-2 compliant services

ICSF is directed to provide FIPS 140-2-compliant services by specifying the FIPSMODE option in the options data set.

The option can be specified as:

- FIPSMODE(YES, FAIL(fail-option)): In this case all applications that call PKCS#11 services must comply with the algorithms and key lengths that the FIPS 140-2 standard dictates.
- FIPSMODE(COMPAT, FAIL(fail-option)): When operating in FIPS compatibility mode, it is expected that further specifications will be made to identify which applications must comply with the FIPS 140-2 standard, and which applications do not need to comply. This is explained in the following sections.

The default is FIPSMODE(NO).

When ICSF starts up with FIPSMODE(YES) or FIPSMODE(COMPAT) specified, ICSF initialization tests that it is running on an IBM System z model type and a version and release of z/OS that supports FIPS. If so, then ICSF will perform a series of cryptographic known answer tests as required by the FIPS 140-2 standard. If any of these initialization self-tests fails, the action that the ICSF initialization process takes depends on the fail-option specified.

Successful completion of the FIPS 140-2 initialization tests is indicated with the following startup message:

```
CSFM015I FIPS 140 SELF CHECKS FOR PKCS11 SERVICES SUCCESSFUL.
```

All applications must be FIPS 140-2 compliant

FIPSMODE has been set to FIPSMODE(YES, FAIL(*fail-option*)), where *fail-option* can be:

- YES, which indicates that ICSF should end abnormally if any of the FIPS 140-2 self tests fail.
- NO specifies that the ICSF initialization process should continue even if one or more of the tests fail. However, z/OS PKCS #11 support will be limited or nonexistent, depending on the test that failed:
  - If ICSF is running on an IBM system z model type or with a version of z/OS that does not support FIPS, most FIPS processing is bypassed. PKCS #11 callable services will be available, but ICSF will not adhere to FIPS 140 restrictions.
  - If a known answer test failed, all ICSF PKCS #11 callable services will be unavailable.

With FIPSMODE(YES), all calls for PKCS#11 services that ICSF receives from an application are checked for compliance with FIPS 140-2 restriction, as indicated in the *ICSF Writing PKCS#11 Applications* book. For instance, it is specified that RSA key size should not be less than 1024 bits.

An example of failure to comply

We again exploited the testpkcs11 test program to show what happens when:

- ICSF has been successfully started with FIPSMODE(YES, FAIL(YES)).
- An application was asking for generation of an RSA key 512 bits long (we modified the value of the CKA_MODULUS_BITS attribute in testpkcs11) for a value of 512, decimal.

Running the modified testpkcs11 program yielded the messages shown in Figure 6-28.

```
KAPPELE @ SC64:/u/kappele/pkcs11>modpkcs11 -t pk.token4
Getting the PKCS11 function list...
Initializing the PKCS11 environment...
Creating the temporary token...
Opening a session...
Generating keys. This may take a while...
Enciphering data...
   C_Encrypt #1 returned:  -1073214466 (0xc0080bfe)
Deleting the temporary token...
```

Figure 6-28 Example of non-compliance to FIPS 140-2

The *ICSF Writing PKCS#11 Applications* book indicates PKCS#11 return codes that are specific to the z/OS implementation. Among these return codes one can find:

```
/* Algorithm or key size is not valid in FIPS mode */
#define CKR_IBM_ICSF_NOT_VALID_FIPS 0xC0080BFE
```

Running in compatibility mode

Compatibility mode can be used when only selected applications should comply with FIPS 140-2.

FIPSMODE has been specified as FIPSMODE(COMPAT,FAIL(fail-option)). Applications can be selected as explained below for strict compliance with FIPS 140-2.

Application user ID and token

An application can be exempted from compliance with FIPS 140-2 on the basis of its RACF user ID and the token that it uses. A profile with a name of the form FIPSEXEMPT.token-label has to be defined in the CRYPTOZ class of RACF resources.

- If a user ID has access authority NONE to the FIPSEXEMPT.token-label resource, ICSF will enforce FIPS 140-2 compliance for that user ID.
- If a user ID has access authority READ to the FIPSEXEMPT.token-label resource, that user ID is exempt from FIPS 140-2 restrictions.

We tested this condition with our modified testpkcs11 program (asking for generation of a 512-bit RSA key) as follows. ICSF was started with the FIPSMODE(COMPAT,FAIL(YES)) options. The profile FIPSEXEMPT.PK.TOKEN4 was defined as follows in RACF (PK.TOKEN4 is the table of the token that we use with testpkcs11):

```
RDEFINE CRYPTOZ FIPSEXEMPT.PK.TOKEN4 UACC(NONE)
```

The the user ID under which testpkcs11 executes was given READ access to the profile:

```
PERMIT FIPSEXEMPT.PK.TOKEN4 CL(CRYPTOZ) ID(KAPPELE) ACC(READ)
```

Although FIPSMODE is not NO and the testpkcs11 program requests a non FIPS 140-2 compliant key size, testpkcs11 completed successfully.

The key that the application uses

When running in FIPS compatibility mode, a PKCS #11 application can, when creating a key, specify that generation and subsequent use of the key must adhere to FIPS 140-2 restrictions. An application specifies this by setting the Boolean attribute CKA_IBM_FIPS140 to TRUE when creating the key.

If an application does this, the FIPS 140-2 restrictions (as outlined in *ICSF Writing PKCS#11 Applications*) will be enforced for the key regardless of any specifications made at the token level using FIPSEXEMPT.token-label resource profiles.

Note: If the installation option FIPSMODE(NO, FAIL(fail-option)), which indicates no FIPS 140-2 compliance for any application, is specified (or defaulted to), an application that sets the Boolean attribute CKA_IBM_FIPS140 to TRUE will fail with return/reason code 8/3071.

Likewise, requests to generate or use a key with CKA_IBM_FIPS140=TRUE will result in a failure return code if ICSF is running on an IBM system z model type or with a version of z/OS that does not support FIPS, although PKCS #11 callable services will be available.

6.7 More on the omnipresent token

Here are facts on the omnipresent token that our experience proved to be useful to know about. Some of them have already been mentioned in previous sections:

- The omnipresent token is always created by ICSF. Its purpose is to enable using PKCS11 services when no other token has been created yet.
- It is available to all applications. That is, USER and SO profiles in the CRYPTOZ class are not checked to determine access to the omnipresent token. All users are considered to have R/W access to the token.
- It is used to store session objects only (that is, objects that do not persist beyond the life of the session).
- It is always mapped to slotID 0, and has an ICSF hard-coded label:

  ```
  SYSTOK-SESSION-ONLY
  ```

- It can be defined as a resource FIPSEXEMPT.token-label, as any other token. The profile is defined using the following RACF command:

  ```
  RDEF CRYPTOZ FIPSEXEMPT.SYSTOK-SESSION-ONLY UACC(NONE)
  ```

- It cannot be deleted. If an application attempts to delete the omnipresent token, the C_InitToken function will fail with a return value of CKR_TOKEN_WRITE_PROTECTED.
- Its contents cannot be visualized using the ICSF Token Browser.

6.8 More on interactive token administration

This section briefly describes how the RACF RACDCERT command and the System SSL gskkyman utility can be used to administer z/OS PKCS#11 tokens.

6.8.1 Examples of RACF token and certificate management options

In many cases, PKCS#11 tokens are used as containers that hold digital certificates and keys. The RACF RACDCERT command or RACF ISPF panels can be used as described below to define and manage certificate-relevant objects in a PKCS#11 token, as opposed to managing the certificate in the RACF database.

Details on the RACF commands that are briefly described here can be found in *z/OS Security Server RACF Command Language Reference*, SA22-7687-15.

The RACDCERT command has the following functions to deal with PKCS#11 token and certificate objects:

- ADDTOKEN defines a new and empty token.
- DELTOKEN deletes an existing token and all of its contents.
- LISTTOKEN displays information about the objects contained in the token.
- BIND binds a RACF digital certificate to an existing PKCS#11 token. It creates, in the token, from the information in the RACF database:
  - The certificate object
  - The public key object
  - The private key object, if the certificate has an associated private key and the BIND USAGE is PERSONAL

Attention: RACF certificates with a private key in the public key data set (PKDS) are not supported for the BIND function. The target private key of the bind operation must be a non-ICSF key.

- UNBIND removes a digital certificate from the specified z/OS PKCS #11 token, without affecting the original certificate and keys in the RACF database.
- IMPORT imports a digital certificate (with its associated private key, if present) from a z/OS PKCS #11 token and adds it to the RACF database.

Important: ICSF must active and a TKDS must be defined for these functions to operate.

Authorizations to use these functions are based on the permissions given to the relevant CRYPTOZ resources.

Because tokens are managed by ICSF, not RACF, other applications can use ICSF functions to change tokens without updating the certificate information in the RACF database. Similarly, RACF changes to digital certificates already bound to a token are not reflected in the token information maintained by ICSF. Therefore, the following restrictions apply:

- Deleting, altering, or renewing a RACF certificate that is bound to a token has no effect on the equivalent token objects managed by ICSF.
- Deleting or altering a certificate object in a token has no effect on the following objects:
  - The equivalent RACF certificate
  - The equivalent certificate objects in other tokens

RACF ISPF panels can also be used, as an alternative to explicitly issuing a RACDCERT command, to manage PKCS#11 tokens. Selecting option 7, digital certificates, key rings, and tokens, from the RACF main panel displays the digital certificate administration main panel (Figure 6-29).

```
 RACF - Digital Certificates and Related Functions
OPTION ===>

  Select one of the following:

      1. Digital Certificate Functions

      2. Key Ring Functions

      3. Certificate Name Filtering Functions

      4. Token Functions
```

Figure 6-29 RACF digital certificate administration main panel

Selecting option 4, Token Functions, displays the panel shown in Figure 6-30.

```
 RACF - Token Functions
OPTION ===>


Enter one of the following:

    1   List all tokens you have permission to read
    2   List an existing token by name
    3   Create a new token
    4   Delete an existing token
    5   Bind a RACF certificate to an existing token
    6   Unbind a RACF certificate from an existing token
    7   Import a token certificate into RACF
```

Figure 6-30 Token Functions main panel

Selecting option 1 in the Token Functions panel displays, in our case, the panel shown in Figure 6-31. These are the tokens that exist and that we have permission to view.

```
 BROWSE - RACF COMMAND OUTPUT----------------------- LINE 00000000 COL 001 080
 COMMAND ===>                                          SCROLL ===> PAGE
******************************** Top of Data *********************************
SYSTOK-SESSION-ONLY

  *** No certificates associated with token ***


******************************* Bottom of Data *******************************
```

Figure 6-31 List tokens that you have permission to view

Notes: Note the following information:

- Only certificates, if any, in tokens are listed. Other objects in tokens do not appear.
- Without a TKDS being defined, only the omnipresent token (SYSTOK-Session-ONLY) is displayed.
- ICSF must be operating for the tokens to be listed. If ICSF is not started, the message shown in Figure 6-32 displays.

```
BROWSE - RACF COMMAND OUTPUT----------------------- LINE 00000000 COL 001 080
 COMMAND ===>                                          SCROLL ===> PAGE
******************************** Top of Data *********************************
Cryptoz processing is not operational. The request is not processed.
******************************* Bottom of Data
```

Figure 6-32 LISTTOKEN and ICSF not started

Creating a PKCS#11 token from RACF

We tried creating a new token using option 3 on the Token Functions panel (Figure 6-33). However, the request failed with ICSF return code 12 and reason code x'BCF'.

```
 RACF - Create a New Token
 COMMAND ===>


     Enter the name of the token to be created:

     PK.TOKEN4_____________________


     Permitted characters are alphanumeric, national (@, #, $) or period (.).
     The first character must be alphabetic or national. Lowercase letters
     are permitted but will be folded to uppercase. Imbedded blanks are
     not allowed.







 Unexpected ICSF CSFPTRC return code x'0000000C' and reason code x'00000BCF'.
The request is not processed.
```

Figure 6-33 Token create and no TKDS available

Looking into the *ICSF Application Programmer's Guide*, we found that the reason code indicates that no TKDS is available.

After restarting ICSF with a properly defined TKDS, the request succeeded. A token list now gives the list shown in Figure 6-34. Note that tokens created by RACF will be indicated to be manufactured by "RACF <racf_fmid>" when displaying the token details.

```
******************************** Top of Data ********************************
PK.TOKEN4

  *** No certificates associated with token ***

SYSTOK-SESSION-ONLY

  *** No certificates associated with token ***


******************************** Bottom of Data ********************************
```

Figure 6-34 Token list

Binding a RACF certificate to the created token

We created a certificate labelled *RACF-created-certificate* in the RACF database using the regular RACF certificate management panels. Figure 6-35 shows the certificate content as displayed by RACF.

```
_  Label:RACF-created-certificate
    Certificate ID:2QfSwdfXxdPF2cHDxmCDmYWBo4WEYIOFmaOJhomDgaOF
    Status:TRUST
    Start Date:2010/05/28 00:00:00
    End Date:  2011/05/28 23:59:59
    Serial Number:00



    Issuer's Name:CN=Patrick Kappeler.T=Redbook co-author.OU=PK Consulti
    ng.O=IBM ITSO.L=Poughkeepsie.C=US
_  Label:RACF-created-certificate
   Certificate ID:2QfSwdfXxdPF2cHDxmCDmYWBo4WEYIOFmaOJhomDgaOF
   Status:TRUST
   Start Date:2010/05/28 00:00:00
   End Date:  2011/05/28 23:59:59
   Serial Number:00



   Issuer's Name:CN=Patrick Kappeler.T=Redbook co-author.OU=PK Consulti
   ng.O=IBM ITSO.L=Poughkeepsie.C=US


   Subject's Name:CN=Patrick Kappeler.T=Redbook co-author.OU=PK Consult
   ing.O=IBM ITSO.L=Poughkeepsie.C=US



   Subject's AltNames:

   EMail: pkappeler at somewhere.fr




Key Usage:HANDSHAKE, DOCSIGN
Private Key Type:Non-ICSF
Private Key Size:1024
```

Figure 6-35 The RACF-created-certificate certificate

We then performed a bind of the certificate into the PK.TOKEN4 PKCS#11 token, using option 5 on the Token Functions panel. Figure 6-36 shows the bind panel.

```
RACF - Bind a RACF Certificate to an Existing Token
 COMMAND ===>

   Identify the RACF certificate that is to be bound:

     Enter the certificate label:
        'RACF-created-certificate'________  (in quotes)
                                                                  More:

     Enter the certificate type:

        Owning user ID: ________  (The default is your user ID)

        OR Select one of the following:
        _ CERTAUTH certificate    OR      _ SITE certificate


   Identify the token to which the certificate should be bound:

     Enter the token name: PK.TOKEN4_____________________


   Determine the usage for the certificate within the token:

     S Don't change the certificate's installed usage (Recommended)
     _ Use the certificate as a PERSONAL certificate
     _ Use the certificate as a CERTAUTH certificate
     _ Use the certificate as a SITE certificate


Determine other certificate attributes:

  S This certificate should be the default certificate for the token
```

Figure 6-36 RACF Bind panel

Note that there is an option, which we did not use, for giving the certificate in the token a different label than its RACF label.

We could then check, using the list tokens function, that the PK.TOKEN4 token now contains a digital certificate (Figure 6-37).

```
 BROWSE - RACF COMMAND OUTPUT----------------------- LINE 00000000 COL 001 080
 COMMAND ===>                                                 SCROLL ===> PAGE
******************************** Top of Data *********************************
PK.TOKEN4

Seq Num  Attributes                   Labels
-------- ---------------------------  ---------------------------------------
00000001 Default: YES   Priv Key: YES  TKDS: RACF-created-certificate
         Usage: PERSONAL Pub Key: YES  RACF: RACF-created-certificate
         Owner: ID(KAPPELE)

SYSTOK-SESSION-ONLY

  *** No certificates associated with token ***


****************************** Bottom of Data ********************************
```

Figure 6-37 Certificate in the token after a RACF bind

More details about the token content can be seen using the ICSF Token Browser (Figure 6-38).

```
-------------------- ICSF Token Management - Token Details --- Row 1 to 2 of 3
 COMMAND ===>                                                  SCROLL ===> PAGE

 Token name: PK.TOKEN4
 Manufacturer: RACF HRF7760
 Model: HCR7770
 Serial Number: 0
 Number of objects: 3

 Select objects to process then press ENTER

 Press END to return to the previous menu.


--------------------------------------------------------------------------------
 _ Object 1         CERTIFICATE   PRIVATE: FALSE      MODIFIABLE: TRUE
                                  DEFAULT: TRUE       CATEGORY: User
    LABEL:          RACF-created-certificate
    SUBJECT:        CN=Patrick Kappeler, TITLE=Redbook co-author, OU=PK Cons...
    ID:             1BDF62E53D5D5830215EC8193C2D719531B0543F
    ISSUER:         CN=Patrick Kappeler, TITLE=Redbook co-author, OU=PK Cons...
    SERIAL NUMBER:  00

 _ Object 2         PUBLIC KEY    PRIVATE: FALSE      MODIFIABLE: TRUE
    LABEL:          RACF-created-certificate
    SUBJECT:        CN=Patrick Kappeler, TITLE=Redbook co-author, OU=PK Cons...
    ID:             1BDF62E53D5D5830215EC8193C2D719531B0543F
    RSA MODULUS:    D19D6D5D456389F57DAA933F00AC200D8C4396271642FD39A4C2E8E4...
    MODULUS BITS:   1024
    USAGE FLAGS:    Enc(T),Verify(T),VerifyR(T),Wrap(T),Derive(T)

_ Object 3         PRIVATE KEY   PRIVATE: TRUE       MODIFIABLE: TRUE
                                  EXTRACTABLE: TRUE   SENSITIVE: FALSE
    LABEL:          RACF-created-certificate
    SUBJECT:        CN=Patrick Kappeler, TITLE=Redbook co-author, OU=PK Cons...
    ID:             1BDF62E53D5D5830215EC8193C2D719531B0543F
    RSA MODULUS:    D19D6D5D456389F57DAA933F00AC200D8C4396271642FD39A4C2E8E4...
    USAGE FLAGS:    Dec(T),Sign(T),SignR(T),Unwrap(T),Derive(F)
```

Figure 6-38 Certificate details using the ICSF Token Browser

Attention: We experienced some confusion while managing the certificate between the certificate serial number and the object sequence number. When a command requires that you refer to the sequence number, that will be sequence number 1 for the above certificate (object 1 in the token), whereas a reference to the certificate serial number will be 00, as indicated in the certificate details above.

The certificate object was selected and the details of its contents were displayed (Figure 6-39).

```
-------------- ICSF Token Management - Certificate Object Details ------------
 COMMAND ===>                                                  SCROLL ===> PAGE
 Object 1      from token label: PK.TOKEN4

 Select an Action:
   1  Process select DER fields(*) using external command.
         Enter UNIX command pathname (see panel help for details):

   2  Modify one or more fields with the new values specified
   3  Delete the entire object
 ------------------------------------------------------------------------------
                                                                        More:     +
    OBJECT CLASS:                    CERTIFICATE
    PRIVATE:                         FALSE
    MODIFIABLE:                      TRUE
    LABEL:                           RACF-created-certificate
                          New value:
    CERTIFICATE TYPE:                X.509
    TRUSTED:                         TRUE
    SUBJECT*:                        CN=Patrick Kappeler, TITLE=Redbook co-au
                                     thor, OU=PK Consulting, O=IBM ITSO, L...
    ID:                              1BDF62E53D5D5830215EC8193C2D719531B0543F
                          New value:
    ISSUER*:                         CN=Patrick Kappeler, TITLE=Redbook co-au
                                     thor, OU=PK Consulting, O=IBM ITSO, L...
    SERIAL NUMBER:                   00
    CERTIFICATE CATEGORY:            User
                          New value: Unspecified   User   Authority   Other
APPLICATION:                        RACF HRF7760
DEFAULT:                            TRUE
                      New value: FALSE
VALUE*:
  3082031B30820284A003020102020100  |0...0...........|
  300D06092A864886F70D010105050030  |0...*.H........0|
  8186310B300906035504061302555331  |..1.0...U....US1|
  153013060355040713 0C506F7567686B |.0...U....Poughk|
  6565707369653111300F060355040A13  |eepsie1.0...U...|
.... more digits not shown here
```

Figure 6-39 Certificate object details

6.8.2 Examples of gskkyman token and certificate management options

gskkyman is a z/OS UNIX utility that is delivered with z/OS System SSL. It can be used to manage digital certificates in UNIX key databases, and, as for the RACF RACDCERT command or ISPF panels, has been extended to manage PKCS#11 tokens and certificates in these tokens.

As for performing the RACF options, ICSF should be running for the gskkyman token management options to operate. The permission to access tokens and their contents is under control of the CRYPTOZ defined profiles.

gskkyman invocation and options

gskkyman is invoked from the z/OS UNIX shell and displays the options list shown in Figure 6-40. Options 11 to 14 pertain to tokens and certificate objects management.

```
KAPPELE @ SC64:/u/kappele>gskkyman

       Database Menu

   1 - Create new database
   2 - Open database
   3 - Change database password
   4 - Change database record length
   5 - Delete database
   6 - Create key parameter file
   7 - Display certificate file (Binary or Base64 ASN.1 DER)

  11 - Create new token
  12 - Delete token
  13 - Manage token
  14 - Manage token from list of tokens

   0 - Exit program

Enter option number:
```

Figure 6-40 gskkyman options

Creating a PKCS#11 token with gskkyman

Figure 6-41 shows a token creation by gskkyman (option 11) that fails because of insufficient authority specified in the RACF CRYPTOZ profiles.

```
Enter option number: 11
Enter token name (press ENTER to return to menu): pk.token5

Unable to create token.
Status 0xcf3b28f0 - Insufficient authority


Press ENTER to continue
```

Figure 6-41 gskkyman token creation failing

We successfully created a token labelled TESTING.TOKEN. Using option 13, we can see the token contents (Figure 6-42) along with the specific management options. Note that tokens created using gskkyman are indicated to be manufactured by z/OS PKCS11 API.

```
Enter option number: 13
Enter token name (press ENTER to return to menu): testing.token

       Token Management Menu

       Token: TESTING.TOKEN

     Manufacturer: z/OS PKCS11 API
     Model: HCR7770
     Flags: x00000509 (INITIALIZED,PROT AUTH PATH,USER PIN INIT,RNG)

   1 - Manage keys and certificates
   2 - Manage certificates
   3 - Manage certificate requests
   4 - Create new certificate request
   5 - Receive requested certificate or a renewal certificate
   6 - Create a self-signed certificate
   7 - Import a certificate
   8 - Import a certificate and a private key
   9 - Show the default key
  10 - Delete Token

   0 - Exit program
```

Figure 6-42 Managing a token with gskkyman

Creating a certificate in the PKCS#11 token

Selecting option 6 above, Create a self-signed certificate, leads to the options shown in Figure 6-43.

```
Enter option number (press ENTER to return to previous menu): 6

       Certificate Type

   1 - CA certificate with 1024-bit RSA key
   2 - CA certificate with 2048-bit RSA key
   5 - User or server certificate with 1024-bit RSA key
   6 - User or server certificate with 2048-bit RSA key

Select certificate type (press ENTER to return to menu): 5

       Signature Digest Type

   1 - SHA-1
   2 - SHA-224
   3 - SHA-256
   4 - SHA-384
   5 - SHA-512

Select digest type (default SHA-1):
```

Figure 6-43 Creating a certificate in the token with gskkyman

The certificate contents can then be defined using the options shown in Figure 6-44. (We only show some of the total options that gskkyman provides for defining the certificate's contents.)

```
Enter label (press ENTER to return to menu): SSSL-created-certificate
Enter subject name for certificate
  Common name (required): Patrick Kappeler
  Organizational unit (optional): PK Consulting
  Organization (required): IBM ITSO
  City/Locality (optional): Poughkeepsie
  State/Province (optional):
  Country/Region (2 characters - required): US
Enter number of days certificate will be valid (default 365):

Enter 1 to specify subject alternate names or 0 to continue: 1

       Subject Alternate Name Type

   1 - Directory name (DN)
   2 - Domain name (DNS)
   3 - E-mail address (SMTP)
   4 - Network address (IP)
   5 - Uniform resource identifier (URI)
```

Figure 6-44 Defining the certificate's contents

The created certificate is labelled SSSL-created-certificate. Its contents can be inspected (Figure 6-45) by selecting the “manage keys and certificates” option for the token.

```
Enter option number (press ENTER to return to previous menu): 1

                        Certificate Information

                Label: SSSL-created-certificate
            Record ID: N/A
      Issuer Record ID: N/A
              Trusted: Yes
              Version: 3
         Serial number: 4bfff87d0006b84a
           Issuer name: Patrick Kappeler
                        PK Consulting
 IBM ITSO
                       Poughkeepsie
                       US
        Subject name: Patrick Kappeler
                       PK Consulting
                       IBM ITSO
                       Poughkeepsie
                       US
      Effective date: 2010/05/28
     Expiration date: 2011/05/28
 Signature algorithm: sha1WithRsaEncryption
    Issuer unique ID: None
   Subject unique ID: None
Public key algorithm: rsaEncryption
     Public key size: 1024
          Public key: 30 81 89 02 81 81 00 B3 5E 75 4A 49 15 62 9A 9D
                      D2 8A 9A C8 0A 2C C3 09 79 15 2C 60 47 15 4A B6
                      92 9A 63 AA 4B 10 61 19 05 0C BA DB CD 57 0A 46
                      A6 CB 0C 31 46 94 C5 76 77 13 63 AF DD 6D EB 92
                      95 38 94 0E 8E FA 3E 7F B3 D1 5B A2 12 C1 B2 94
                      23 8C CD 93 1C D9 1F 82 B8 AB A6 2F 52 27 E7 C4
                      A7 5F 14 34 F8 85 64 9A 9B 12 71 61 89 2B 69 01
                      3F 6F F0 61 94 36 43 DA EF 44 49 E2 B8 73 F7 55
                      5B AE 0F EF 33 34 7F 02 03 01 00 01
Number of extensions: 5
 Number of extensions: 5
```

Figure 6-45 Contents of the certificate created in the token

The ICSF Token Browser can also be used to inspect the certificate (Figure 6-46). Note that both private and public keys were created in the token.

```
 -------------------- ICSF Token Management - Token Details --- Row 1 to 2 of 3
 COMMAND ===>                                                  SCROLL ===> PAGE

 Token name: TESTING.TOKEN
 Manufacturer: z/OS PKCS11 API
 Model: HCR7770
 Serial Number: 0
 Number of objects: 3

 Select objects to process then press ENTER

 Press END to return to the previous menu.


--------------------------------------------------------------------------------
 _ Object 1         CERTIFICATE   PRIVATE: FALSE      MODIFIABLE: TRUE
                                  DEFAULT: FALSE      CATEGORY: User
    LABEL:          SSSL-created-certificate
    SUBJECT:        CN=Patrick Kappeler, OU=PK Consulting, O=IBM ITSO, L=Pou...
    ID:             788B883073EC445BE872C56FD51C19610C41B536
    ISSUER:         CN=Patrick Kappeler, OU=PK Consulting, O=IBM ITSO, L=Pou...
    SERIAL NUMBER:  4BFFF87D0006B84A

 _ Object 2         PUBLIC KEY    PRIVATE: FALSE      MODIFIABLE: TRUE
    LABEL:          SSSL-created-certificate
    SUBJECT:        CN=Patrick Kappeler, OU=PK Consulting, O=IBM ITSO, L=Pou...
    ID:             788B883073EC445BE872C56FD51C19610C41B536
    RSA MODULUS:    B35E754A4915629A9DD28A9AC80A2CC30979152C6047154AB6929A63...
    MODULUS BITS:   1024
    USAGE FLAGS:    Enc(T),Verify(T),VerifyR(T),Wrap(T),Derive(F)

_ Object 3         PRIVATE KEY   PRIVATE: TRUE       MODIFIABLE: TRUE
                                  EXTRACTABLE: TRUE   SENSITIVE: FALSE
    LABEL:          SSSL-created-certificate
    SUBJECT:        CN=Patrick Kappeler, OU=PK Consulting, O=IBM ITSO, L=Pou...
    ID:             788B883073EC445BE872C56FD51C19610C41B536
    RSA MODULUS:    B35E754A4915629A9DD28A9AC80A2CC30979152C6047154AB6929A63...
    USAGE FLAGS:    Dec(T),Sign(T),SignR(T),Unwrap(T),Derive(F)
```

Figure 6-46 Inspecting the certificate contents with the ICSF Token Browser

6.8.3 Moving the certificate from PKCS#11 token to the RACF database

The RACDCERT IMPORT command or the RACF panels can be used to import the certificate that we created in the token with gskkyman into the RACF database.

7

Trusted Key Entry

In this chapter we provide an overview of the Trusted Key Entry (TKE) workstation Version 6.0. Version 6.0 is the follow-on to the TKE V5.3 level. Version 6.0 is the level required to support Crypto Express3 Cards (CEX3C) on z10. TKE V6.0 also works on the same workstation as TKEV5.3 using the IBM proprietary operating system and the 4764 cryptographic adapter.

Several other IBM Redbooks address, from a broader perspective, cryptographic services implementation in the System z, as well as the use of the TKE workstation in this context. We suggest that readers who are unfamiliar with the TKE concept and implementation see *z9-109 Crypto and TKE V5 Update,* SG24-7123, for further background information.

For further information about TKE V6.0, refer to *z/OS Cryptographic Service, Trusted Key Entry PCIX Workstation User's Guide,* SA23-2211.

7.1 Introduction to the TKE

The TKE workstation is a priced optional feature used for management of zSeries, System z9, and System z10 secure coprocessors in an installation. Secure coprocessors operate with a master key that resides inside the coprocessor itself.

These secure coprocessors use application keys that are protected by being encrypted with the master key. The application keys are only decrypted inside the coprocessor secure enclosure.

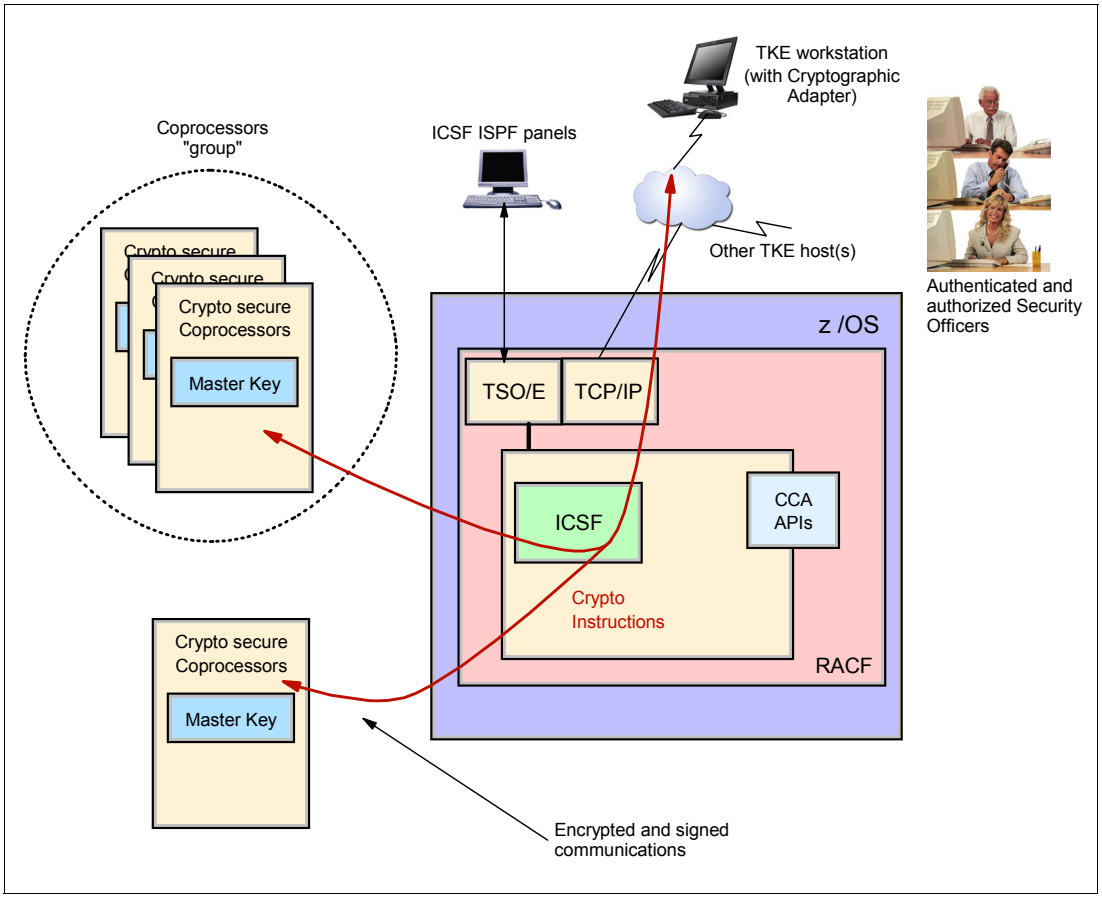


Figure 7-1 A schematic view of the TKE workstation implementation

7.1.1 What can be done with a TKE

TKE provides secure remote key management capabilities for the coprocessors' master keys and operational keys by communicating directly with the coprocessors, using encrypted and digitally signed communications. The coprocessors execute commands entered at the workstation by security officers who have been authenticated using RSA public cryptography, and who are authorized to issue TKE commands.

The TKE workstation communicates, via TCP/IP, with one or several ICSF instances known as the *TKE hosts*. These ICSF instances act just like a TCP/IP listener, and they relay the communication to the coprocessors themselves. A single TKE can be used to manage secure coprocessors in the systems to which it has TCP/IP connectivity. Secure coprocessors can also be logically grouped at the TKE level so that the TKE application will automatically propagate a single command to all coprocessor members of the group.

7.1.2 Why TKE is necessary

If you are aiming for a high level of data confidentiality and integrity, you are likely to install the TKE workstation. It allows you to keep your cryptographic keys secret and protected from unauthorized access.

The TKE provides the only secure method to enter master key values into Cryptographic coprcessors installed on System z. The other method to enter the master key values is to use ICSF panels where the master key values are in clear text.

TKE also provides protected storage for the master key and operational key parts by using the smart card feature on the TKE. The key parts can only be sent to the Cryptographic coprocessor or copied into a second smart card inside the same zone. The TKE enables the use of dual custody through roles that enable high-security processes when using the TKE.

7.1.3 Smart card support

When properly installed and administered, using smart cards with the TKE workstation provides a high level of security. Smart card support gives the user the ability to keep all key parts, signature keys, and TKE crypto adapter logon keys from ever appearing in clear text.

Smart card support requires:

- TKE V4.2 or later code
- TKE smart card readers with smart cards
- TKE workstation with an IBM Cryptographic Adapter Card
- TKE workstation initialized for the smart card support option

Note: The 4764 card is certified at FIPS 140-2 Level 4 for the hardware, segment 0 and segment 1. Segments 2 and 3 are not certified. For TKE V6.0, the 3.50 level or later of the licensed internal code (LIC) is required for segments 2 and 3.

The TKE workstation with smart card support:

- Stores ICSF (host) key parts (specifically, master and operational key parts on TKE smart cards)
- Stores TKE crypto adapter workstation master key parts on TKE smart cards
- Generates, stores, and uses a TKE authority signature key on TKE smart cards
- Generates, stores, and uses a TKE crypto adapter logon key on TKE smart cards

Zone concept

The smart card feature on TKE6.0 works in a functionally similar way as that on TKE5.x. Support is designed around the concept of a zone. This is done to ensure the secure transfer of key parts. The members of a zone are:

- CA smart card
- TKE cryptographic adapter cards
- TKE smart cards

A member of a zone is referred to as an entity. Entities have to be in the same zone before they can exchange key information. A zone will be created using a CA smart card. A TKE smart card and TKE Crypto adapter will be enrolled to the zone.

The zone ID is checked only when exchanging key parts between TKE smart cards or a TKE smart card and TKE Workstation Crypto Adapter.

7.1.4 Multiple TKEs

It might be desirable to have multiple zones, especially if you have multiple TKE workstations. In fact, we suggest that separate zones be created for testing and production systems. This prevents keys from getting intermixed.

Note that entities can only be a member of one zone at any given time.

Figure 7-2 shows multiple zones for a production and test system. The production system has a remote TKE workstation enrolled. The test system does not. There are separate CA smart cards associated with each zone.

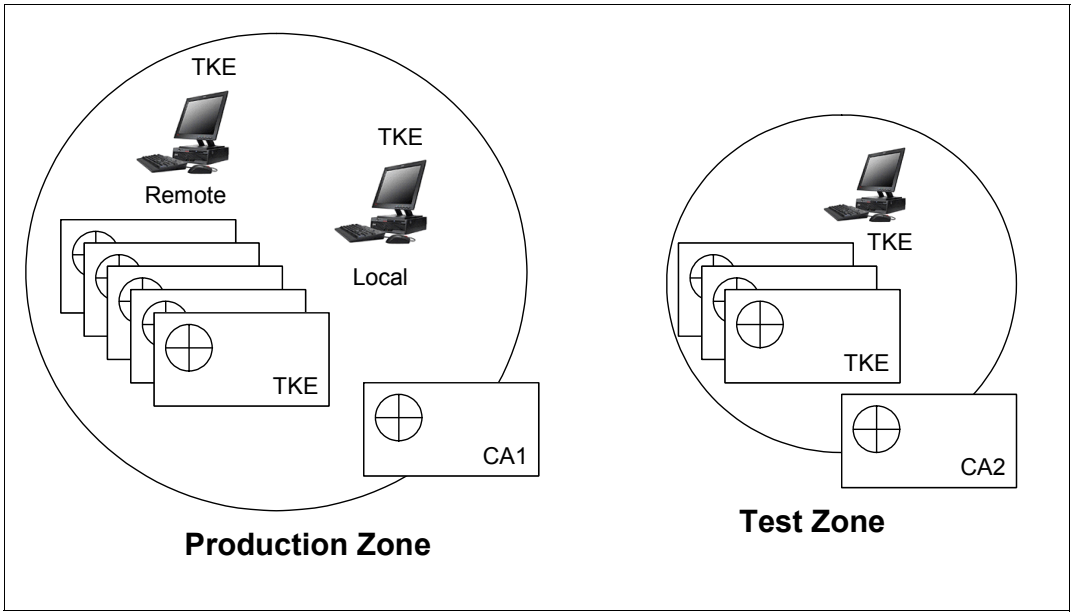


Figure 7-2 Multiple TKEs and multiple zones

7.1.5 Roles and profiles

TKE operations rely on the capability to use the imbedded 4764 cryptographic adapter. The 4764 implements an access control mechanism that uses the *roles* and *profiles* concept (Figure 7-3). When necessary, these roles and profiles are defined by the TKE administrator using the Crypto Node Management (CNM) utility and according to the customer's security policy. This utility, included in TKE code, is started from the Trusted Key Entry window under Applications and requires proper user authentication and corresponding privileges to access the administrative functions.

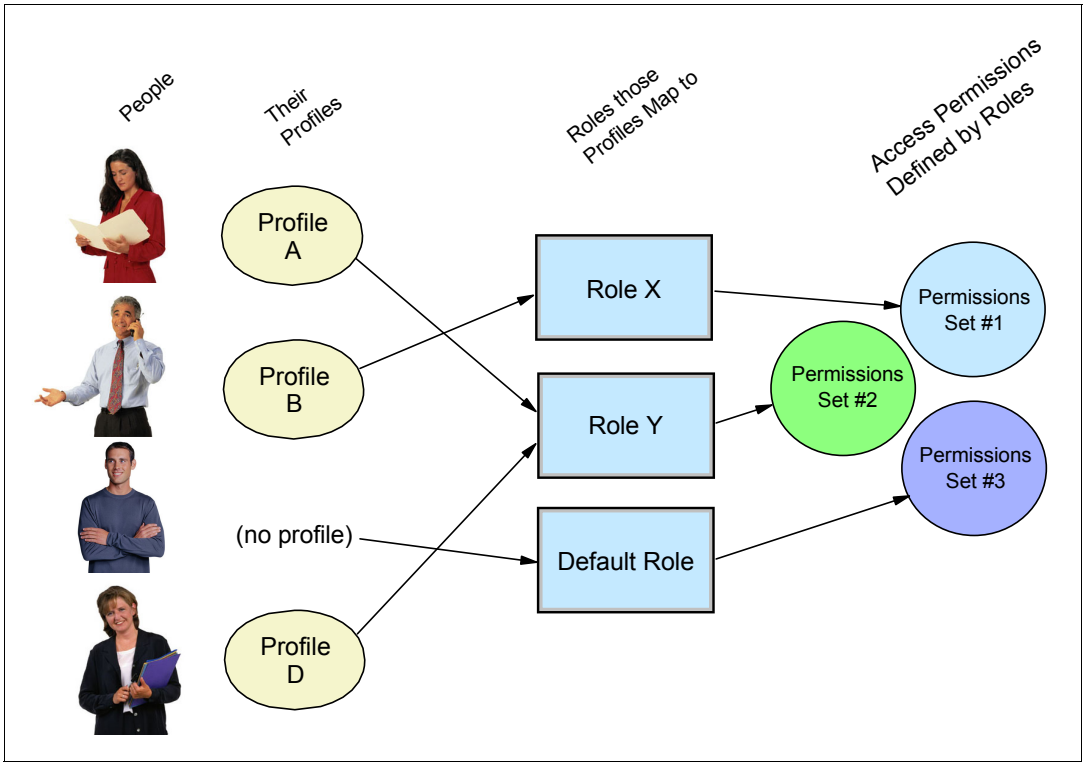


Figure 7-3 Role-based access control

Role

A role defines a class of TKE users who can execute a set of the 4764 cryptographic adapter operations. When creating or changing a role, the TKE administrator will define the TKE 4764 Crypto card commands that will be authorized for users who are mapped to this role.

The TKE 4764 has a DEFAULT role. Use of the DEFAULT role does not require a user profile. Any user can use the services permitted by the DEFAULT role without logging onto or being authenticated by the coprocessor. However, the DEFAULT role is limited in the operations that it is allowed to perform. Most of these operations are service management related.

Note: Predefined roles are set up during TKE initialization, and you should not to change these roles, as that might cause the TKE application processes to fail.

Profile

A profile defines a specific user. It contains a user name and passphrase information, with the Passphrase initialization option. A profile is mapped to only one role, though as many profiles as needed can be created and mapped to the same role.

After logging on under a profile, the TKE user can perform only the commands authorized by the role mapped to the profile. Validity dates might also be defined on a role basis to enforce specific security policies.

Predefined profiles are set up during TKE initialization (passphrase or smart card initialization). Passphrase profiles will have a default known passphrase. We suggest that you change this passphrase after TKE setup to ensure full access security.

The TKE administrator must create all user profiles required by the security policy in effect.

Important: The CEX2/CEX3 coprocessor in the System z uses a similar access control concept that involves roles and authorities. However, these roles and authorities must not be confused with TKE 4764 roles and profiles.

User logon to the TKE

TKE requires a logon for most of the Trusted Key Entry applications. The user logs on to the TKE providing a user ID and a passphrase. The user ID is a TKE 4764 user profile, and if the passphrase is correct the TKE user is logged in and can invoke this role's authorized 4764 functions. With the smart card option you use smart card and PIN instead of user ID and passphrase. The logon process is similar in the two cases.

An unsuccessful or anonymous logon does not give access to those Trusted Key Entry applications where logon is required.

Note: After three incorrect passphrase tries, TKE will lock the profile and then it cannot be used before it is reset using CNM. After three incorrect PINs on the smart card TKE will lock the card, making it unusable. The card must be unlocked on the smart card utility program (SCUP) with the CA smart card.

A successful logon in the TKEUSER role enables communication with the System z coprocessors for administrative purposes if the correct authority signature key is used.

7.1.6 Group logon

As explained in the previous chapter, access control to the TKE is achieved by defining 4764 user profiles with a user ID and an authentication passphrase. The passphrase can be replaced by authentication data provided in a smart card. Logging on is required to administer System z coprocessors or to use Trusted Key Entry applications such as CNM and SCUP.

Since TKE V4.2, the group logon feature has been available to provide multi-access controls to get logged on to the TKE 4764 cryptographic adapter. This is based on the use of a group profile, which requires individual authentication of each of the users belonging to the group.

A group profile has the same structure as standard profiles but uses a newly defined authentication mechanism, which is a list of names of other profiles. The group logon can be used with the passphrase mechanism or with the smart card feature using public key authentication. The logon request references the group profile, and then proceeds with the authentication of the individual members of the group.

A group can have from one to 10 members, although not all members have to log on at the same time. All members of a group must use the same authentication mechanism (passphrase or smart card). That is, a group is either:

- One to 10 users who are authenticated by passphrase
- One to 10 users who are authenticated by smart card

Note: A group cannot be a member of another group.

Important: It is important not to mix the passphrase and the smart card authentication method. You should choose which to use and initialize the TKE using either authentication method.

7.1.7 How TKE communicates with System z, Crypto Express, z/OS, and ICSF

In this section we discuss how TKE communicates with System z, Crypto Express, z/OS, and ICSF.

TKE Listener

The TKE Host Transaction Program (TKE HTP) is the host-based part of Trusted Key Entry. It forms the interface between the TKE workstation and the crypto modules.

The TKE HTP (server) needs to be started before a TKE workstation (client) can communicate with the host crypto modules. The TKE HTP consists of a started procedure (CSFTTCP) and a REXX clist (CSFTHTP3), which calls a module (CSFTTKE) that does SAF authorization checking to make sure that no unauthorized clients get to the TKE HTP server.

To run the new TKE Host Transaction program, the CSFTTKE module must be added to the authorized command list in IKJTSOxx on the system where the TKE HTP server will be started.

Perform these steps to install the server:

1. Update the authorized commands list in the TSO commands and programs member, IKJTSOxx, in the SYS1.PARMLIB data set.
2. Set up system security.

   To protect module CSFTTKE from unauthorized users, you must protect it using RACF. For example, permit the user ID or group assigned to the CSFTTCP started task to the CSFTTKE profile in the FACILITY class.

   It is strongly suggested that the module (CSFTTKE) is also protected. This can be done using the APPL class to control which users can use the application when they enter the system.

   If you do not have a generic user ID associated with all started procedures, you can associate a user ID to the CSFTTCP proc by issuing a RACF RDEFINE command for a STARTED class profile.
3. The TKE Host Transaction program must be started before you can log on to the host from TKE. A sample startup procedure is shipped in SYS1.SAMPLIB(CSFTTCP). Copy this procedure to your proclib data set and customize it for your installation.

Note: TKE Host Transaction program uses the TKECM file to store descriptions about Host Cryptographic Cards, host domains, and authorities. If upgrading from an older legacy machine prior to a z990 and upgrading to TKE6.0, you must delete or rename the existing TKECM data set. The TKECM data set prior to TKE5.0 is not compatible with a z990, z890, z9 EC, z9 BC, z10 EC, or z10 BC system TKECM data set.

7.2 HIstory of TKE

The ancestor of the TKE was the frame-mounted manual-control panel provided with the Integrated Cryptographic Facility (ICRF) feature in the bipolar ES9000 systems. This panel was used by a security officer standing at the system, entering the master key (there was only one master key per domain at that time) or operational keys parts. The panel was physically wired to the crypto TCM using a very expensive tamper-resistant cable. The cable consisted of parallel wires and several layers of metal shielding to protect transmission from eavesdropping. The cable was also protected against physical intrusion, such as breaking or cutting the cable, which triggered the tamper indicator when detected. Protection was also provided against attacks utilizing low temperature and ionizing radiation.

With the advent of the hardware cryptography on the 9672 CMOS systems, security officers still had to enter master keys and operational keys parts, but the concept of the tamper-resistant cable was revisited and it became a cryptographically secured network connection between a special workstation and the cryptographic coprocessors called Public Key Secure Cable (PKSC). The network connection was relayed to the coprocessors by only one instance of ICSF in the system (meaning that other instances of ICSF can run in other LPARs, but are not involved in communicating with the TKE).

Note: A single TKE can be used to administer the crypto coprocessors in several physical systems if it has TCP/IP connectivity to these systems, and one instance of ICSF and of the TCP/IP listener is running in each of them. A single TKE can also be used to administer all of the domains in the coprocessors of a single system if one instance of ICSF and of the TCP/IP listener runs in one LPAR.

7.2.1 TKE V1/V2 (1997 to 2000)

The TKE V1 and V2 workstation was OS/2-based and communicated with ICSF using VTAM® APPC over an SNA network. Two LAN attachments (token-ring or Ethernet) and a wide area connection (a modem attachment) were offered.

The TKE workstation used a 4755 cryptographic adapter internally to sign and encrypt communications with the coprocessors and, optionally, could be attached to an IBM smart card reader attachment, the IBM 4754 Security Interface Unit with IBM proprietary Personal smart card (PSC) technology. The TKE was used to administer the CCF, the only coprocessor that was available then.

7.2.2 TKE V3 (2000)

With the introduction of the PCICC card for the 9672 G5/G6 system, the TKE workstation code was enhanced for remote administration of the PCICC coprocessors along with CCF administration. The TKE also evolved internally:

- It used an IBM 4758 cryptographic adapter.

- There was no longer any support for smart cards.
- The only supported network was TCP/IP with a token-ring LAN attachment (FC 0866) or Ethernet attachment (FC 0869). The wide area connection was no longer available.
- The TKE V3.1 also implemented the concept of hardware access control points in the PCI coprocessors.

7.2.3 TKE V4 (2004)

With the introduction of the PCIXCC card for the z990 system, the TKE workstation code was enhanced for remote administration of the PCIXCC coprocessors along with CCF administration. The TKE also evolved internally:

- TKE used an IBM 4758 cryptographic adapter.
- There was new support for smart card feature (TKEV4.2).
- The only supported network was TCP/IP with a token-ring LAN attachment (FC 0866) or Ethernet attachment (FC 0869).

7.2.4 TKE V5 (2005)

With the introduction of the PCIXCC card for the z990 system, the TKE workstation code has been enhanced for remote administration of the PCIXCC coprocessors along with CCF administration. The TKE also evolved internally:

- TKE used an IBM 4764 cryptographic adapter.
- There was a new workstation and a new imbedded operating system.
- The only supported network was TCP/IP with Ethernet attachment.
- The smart card feature introduced on TKEV4.2 was also supported.

7.2.5 Table of TKE levels and capabilities

Table 7-1 is a brief reference as to where the latest TKE release can be found with related connections and feature highlights.

Table 7-1 Table of the TKE features and capabilities

TKE version	Supported mainframe	Highlights	Crypto card	Operating system	DVD-RAM	Diskette
TKEV4.2	z990, z890, z900, z800	Smart card support, group logon	4758	OS/2	N	R/W
TKEV5.0, 5.1, 5.2	z9, z990, z890, z900, z800	New workstation, new operating system, new Crypto card, operational key enhancements, IvP6 (V5.2)	4764	Embedded	Y	Read-only (V5.2)
TKEV5.3	z10, z9, z990, z890, z900, z800	New smart card readers and cards (readers with USB connect), ADMIN, AUDITOR, AES master keys and AES secure keys, print screen utility	4764	Embedded	Y	Read-only

TKE version	Supported mainframe	Highlights	Crypto card	Operating system	DVD-RAM	Diskette
TKEV6.0	z10, z9, z990, z890	CEX3C support, migrate configuration data utility, zone and signature key strength enhancements	4764	Embedded	Y	Read-only

7.3 The latest TKE, Version 6.0

We now move on to describe the features of TKE Version 6.0.

7.3.1 Hardware

The basic workstation hardware consist of the following components:

- The TKE workstation itself (FC 0840) with TKE6.0 LIC code (FC 0858).

  Note: If you are migrating from TKE5.x LIC code to TKE6.0 LIC code level, the same TKE5.x workstation hardware can be used with TKE6.0 LIC.

- TKE uses the IBM 4764 Cryptographic Adapter.

Optional hardware

Also available with a TKE 6.0 workstation are:

- Feature 0885: Two OmniKey smart card readers and 20 smart cards
- Feature 0884: Ten smart cards

Note: Note the following information:

- OmniKey smart card readers require TKE 5.3 or TKE V6.0 code - FC 0854 with the November 2008 or later licensed internal code (LIC).
- Kobil smart card readers can no longer be ordered but can be used with TKE V4.2 or later.
- Older smart cards must be reinitialized on TKE V5.3 or later to be able to store AES master keys and AES operational keys.
- Older smart cards must be reinitialized on TKE 6.0 to be able to store a 2048-bit signature key and support 2048-bit certificate strength.
- In general, we suggest that you use smart cards that support 2048-bit keys. Note that the following combinations of smart cards and smart card readers do not support 2048-bit RSA keys:
  - Kobil Readers and DataKey smart card
  - Kobil Readers and NXP smart card
  - OmniKey Readers and DataKey smart card

USB connectors/memory support

USB connectors in the front of the TKE workstation are used by the OmniKey smart card readers.

USB memory can be used to store print screen pictures copying from the TKE hard disk storage.

7.3.2 Domain grouping

Host image profiles for logical partitions must be correctly configured in order to use the TKE workstation to manage keys and perform other operations. The host Support Element or Hardware Management Console is used to set and change the configuration.

When customizing an image profile using the support element, four fields are specified:

- Usage domain index: The domain associated with the logical partition.
- Control domain index: The set of domains that can be managed from this logical partition. It must include the usage domain index value for this logical partition and usage domain indexes for all those logical partitions that the TKE workstation can manage through this logical partition.
- PCI Cryptographic Candidate List: The set of cryptographic coprocessors that the logical partition can access but that are not brought online when the logical partition is activated.
- PCI Cryptographic Online List: The set of cryptographic coprocessors that will be brought online when the logical partition is activated.

There is no specific field to identify a logical partition as a TKE host when you are customizing image profiles. You must decide which logical partition will be the TKE host and set up the control domain index appropriately. The control domain index for this partition must include the usage domain index values for all logical partitions that the TKE workstation will control, and the PCI Cryptographic Candidate List for this partition must include all entries in the PCI Cryptographic Candidate Lists for the logical partitions that the TKE workstation will control. The control domain index must also include the usage domain index value for the TKE host partition itself.

Planning domains (old and new method)

Using grouping with the TKE has a definite need. Grouping allows the definition of host settings, such as roles and authorities, and also master key values, with less work and therefore with less errors. Grouping is not mandatory, but it is useful when more than one LPAR will use the same settings or master key values on their usage domains.

Prior to TKE V6.0, Cryptographic Cards could be grouped only at the card level. The target for this method is that all cards in the same group will have the same definitions at the card level. With this method the master key values are stored in the same domain on all cards in the group. This is not complicated to define with a few LPARs on a single mainframe, but it requires careful planning if there are several mainframes and on each of them several LPARs.

Below are a few points that will affect how to plan grouping:

- How many different master key values are needed?
- Can the production and test environments be separated or are they forced to be separated?
- Does the LPAR have a standby or alternate location on the same or a different mainframe?
- Can the master key values be entered into the standby LPARs in advance?
- Do the production and test environments need different roles and authorities, such as having different master key holder personnel?

- Usually sysplex members that use ICSF will use shared CKDS and PKDS data sets, thus requiring the same master key values.
- Which LPARs can be used as TKE hosts and is there a backup?

All of the points above are much easier when creating a new setup or migrating to a new mainframe generation with new Cryptographic cards. Our experience shows that in most cases a new TKE is introduced on an existing environment to enhance security and speed by using TKE for the master key entry instead of the TSO panels.

Going through options for the grouping, most of the need is covered using Crypto module grouping, though it is easy to create overlaps depending of the complexity of the installation.

Domain grouping implementation

TKE V6.0 has a new feature called domain grouping, which will help especially in an environment were the existing ICSF usage domain setup is difficult or impossible to change.

We suggest not intermixing Crypto module group and domain group definition targets because an update on either side could override definitions made earlier. A Crypto card has host roles and authorities at the card level whatever grouping method is used. The master and operational keys, domain zeroize, control and access are all at the domain level.

Doing careful planning work is also important with this new feature. The following set of foils introduces how to define domain grouping on the TKE V6.0. The new feature is located on the TKE application main window. Right-click in the Domain groups window and select **Create Group** (Figure 7-4).

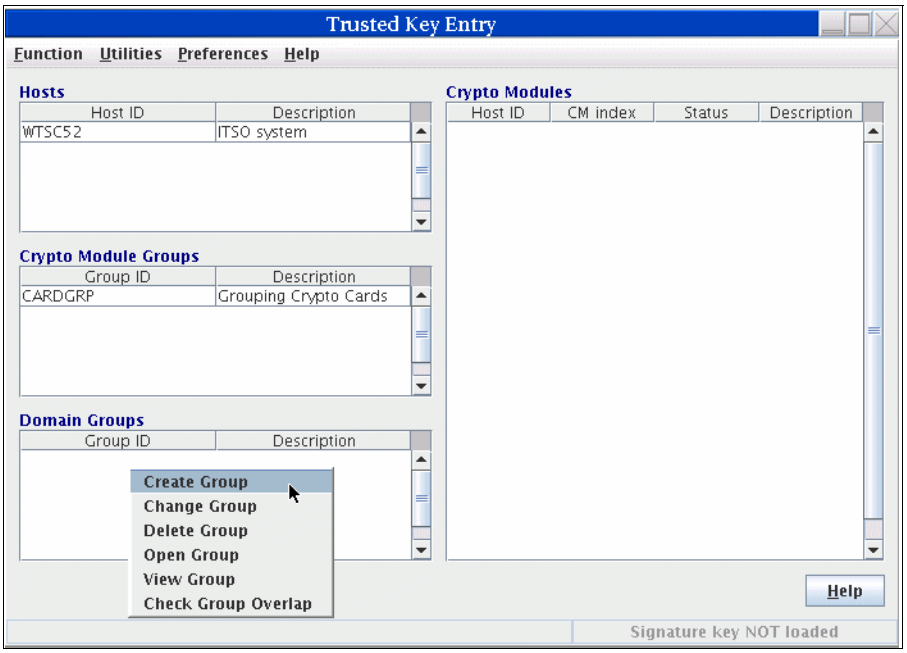


Figure 7-4 Creating a Domain group

A new window displays where you can fill in group identification and an optional description. The list shows the defined hosts (in our case only one). Select the host (Figure 7-5). A group can have domains from several hosts and several Crypto cards.

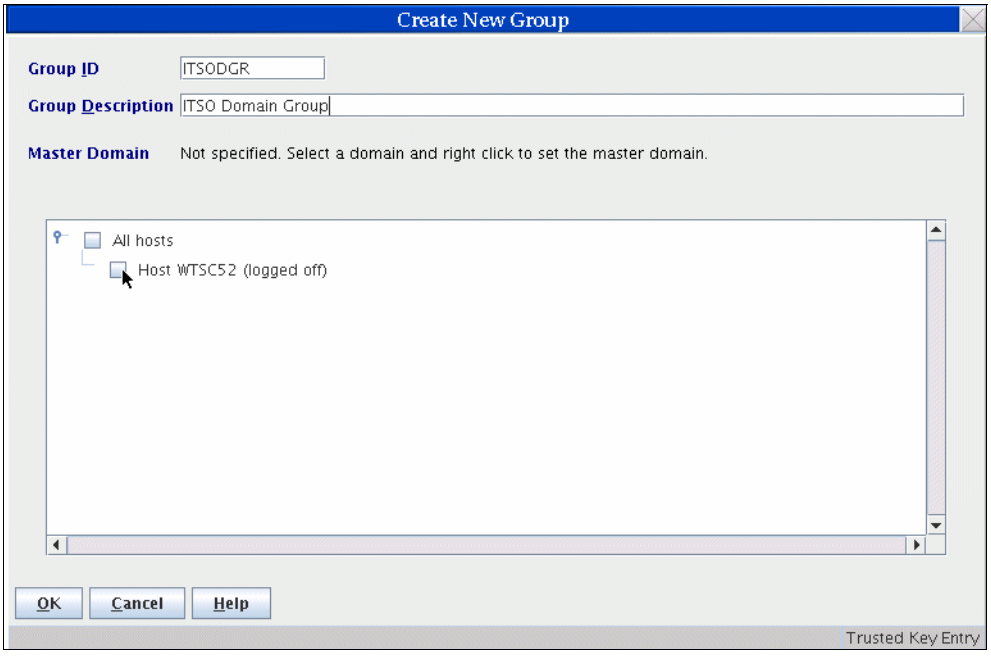


Figure 7-5 Select a host for the domain group

A prompt displays to create a connection to the host where the TKE listener is running. Enter your user ID and password (as known to the security system, such as RACF).

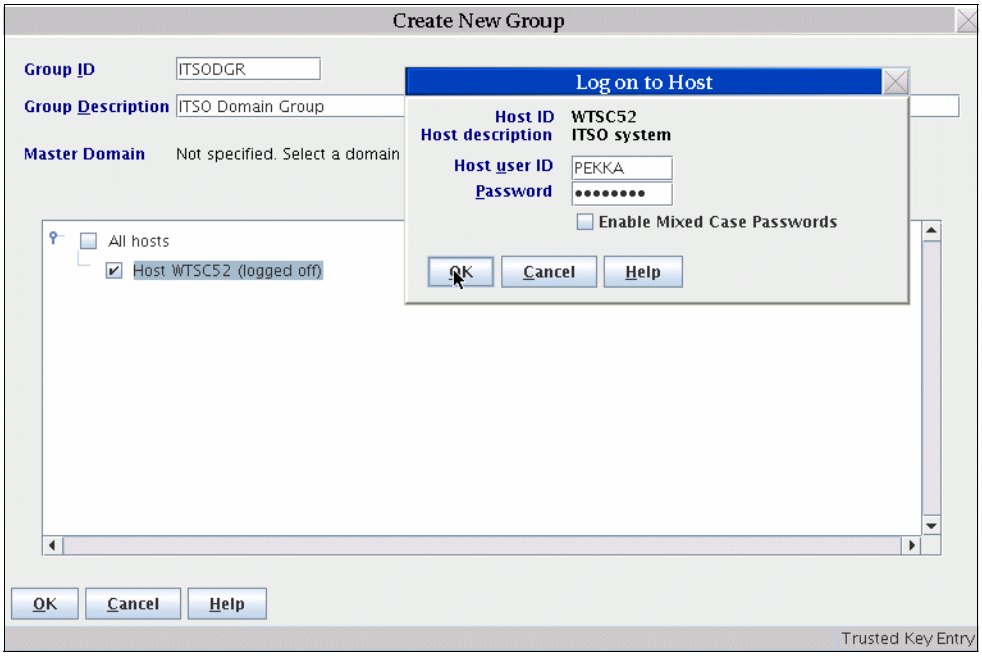


Figure 7-6 Log on to the host to access Crypto card information

Expand the wanted crypto module by clicking the marker on the left of the check box. Select the target domains by clicking the check boxes. It is also possible to select all domains on the Crypto module by clicking the check box next to the Crypto Module (Figure 7-7).

When selecting only part of the domains for the group, a gray area surrounds higher level selection check boxes (Figure 7-7).

Note: Selecting all domains on a Crypto Module is valid only for those domains that are defined on the control domain list on the LPAR Cryptographic settings.

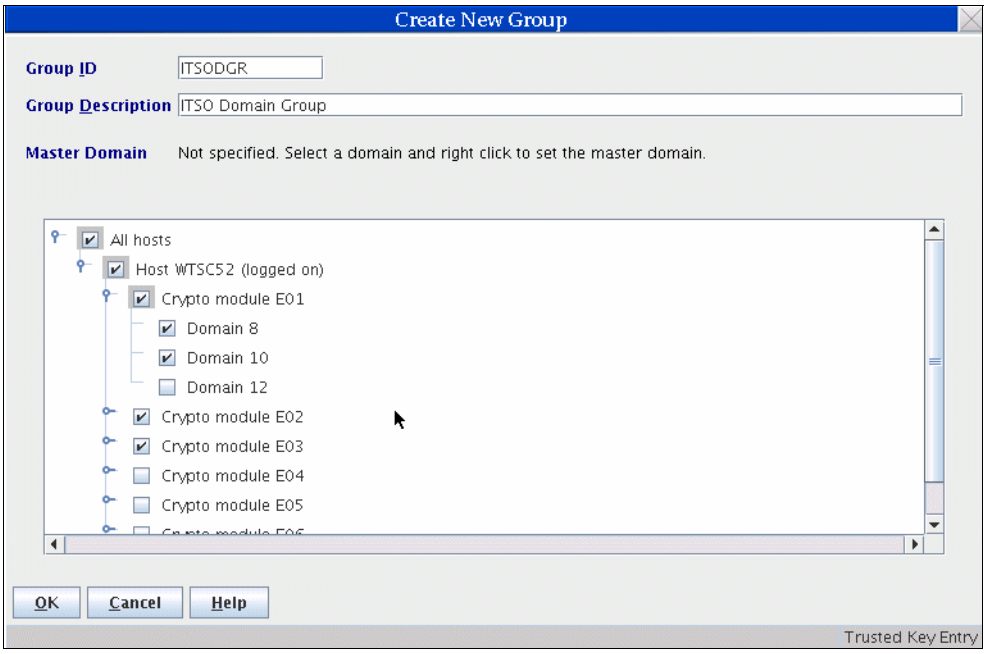


Figure 7-7 Select which domains are part of the group

One of the domains must be the master domain for the group,.Select the desired domain from the list of selected domains. Right-click and select **Make this the master domain** (Figure 7-8). The master domain will be used when group information such as the master key hash patterns are viewed and also on the group compare process.

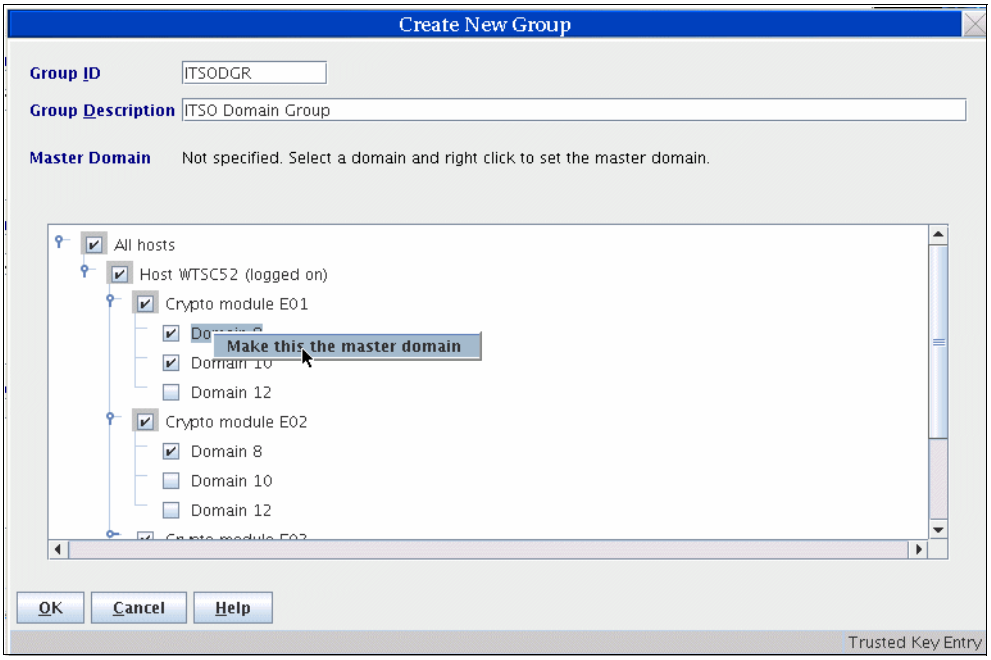


Figure 7-8 Select master domain for the group

To view the defined domain groups, right-click the desired domain group and select the **View Group** option (Figure 7-9).

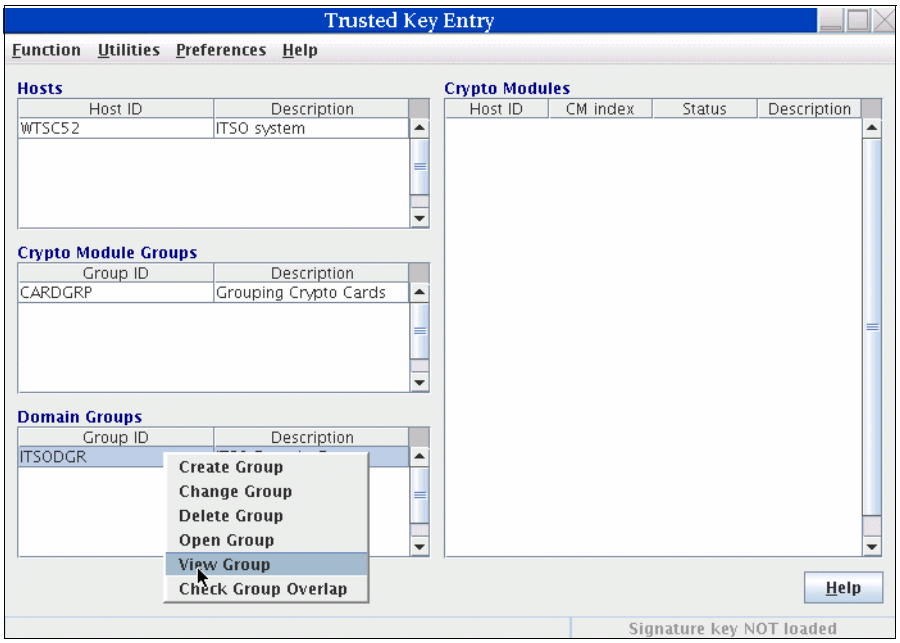


Figure 7-9 View the created domain group

A new window displays showing the content of the domain group (Figure 7-10). The domain View Group windows show which domains are selected to be in the group and also which of the domains is the master domain. This information also can be saved to a file for documentation purposes. To do this, click **Copy to File** (Figure 7-10).

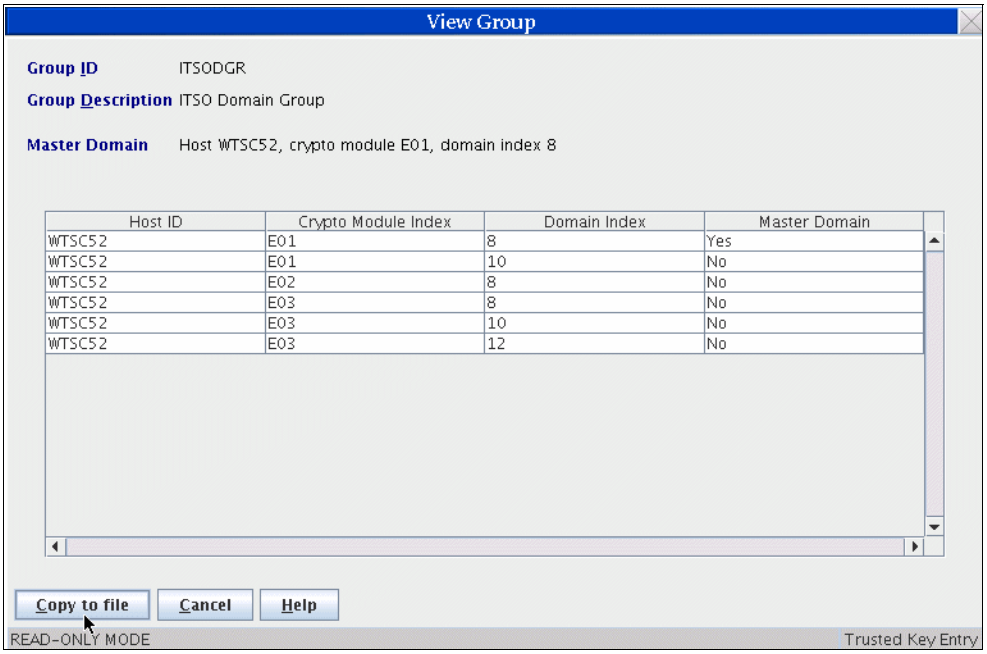


Figure 7-10 Save the content of the selected domain group

The information can be saved on the TKE hard disk to the TKE Data Directory (Figure 7-11) or to the CD/DVD.

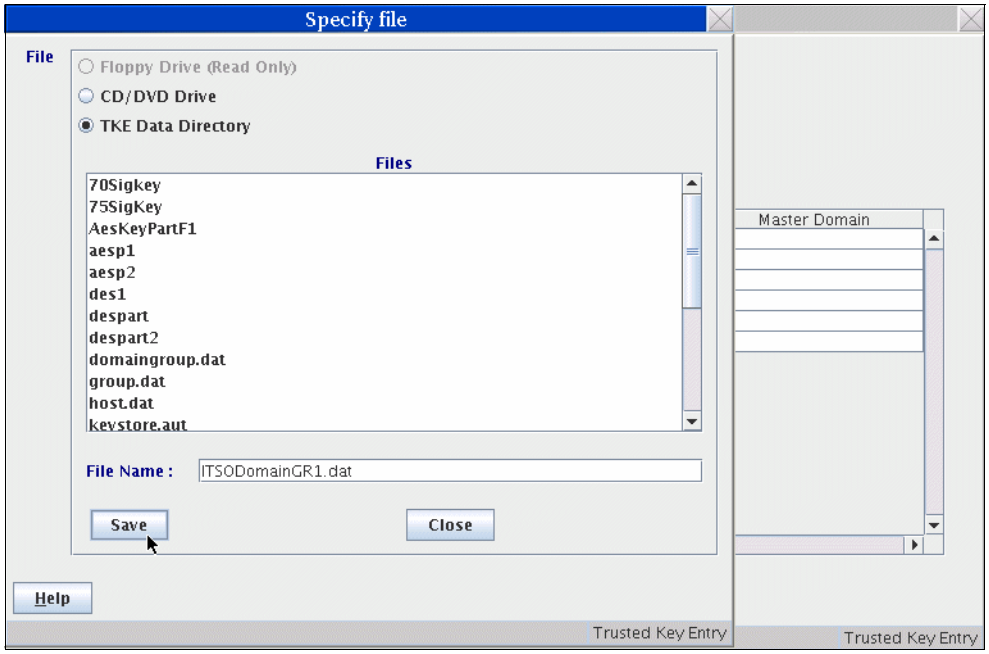


Figure 7-11 Store the domain group information content in a file in the TKE data directory

For testing purposes we define a second domain group (Figure 7-12).

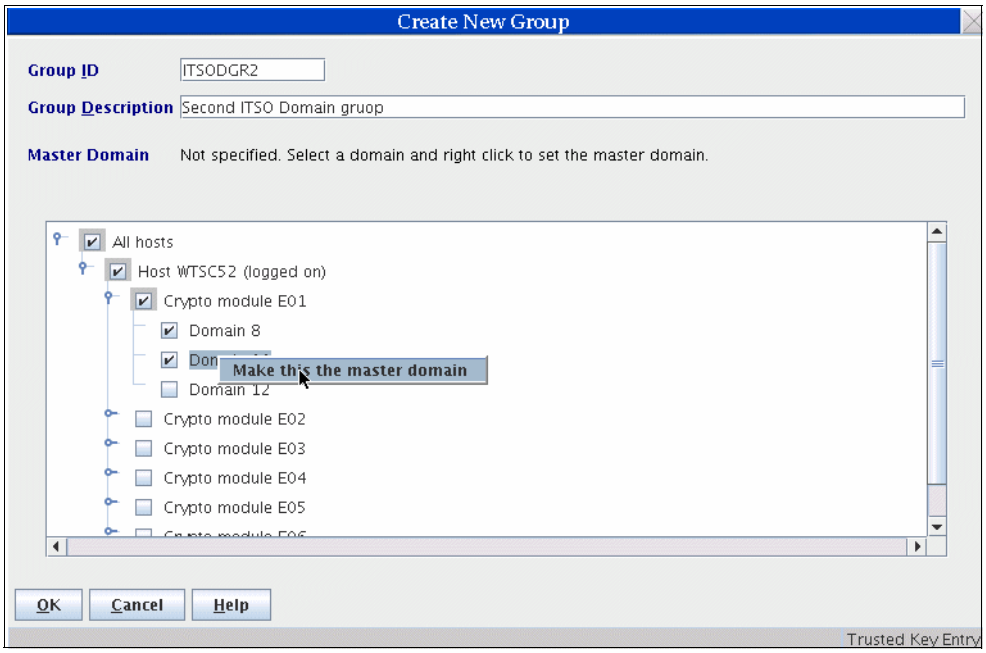


Figure 7-12 A second domain group is created and master domain is set

With several domain groups it is easy to make a mistake and define the same domain to be a member in several groups. This causes problems because the updates will overwrite each other and can create unpredictable results.

To avoid overlapping, right-click in the Domain Groups box and select **Check Group Overlap** (Figure 7-13).

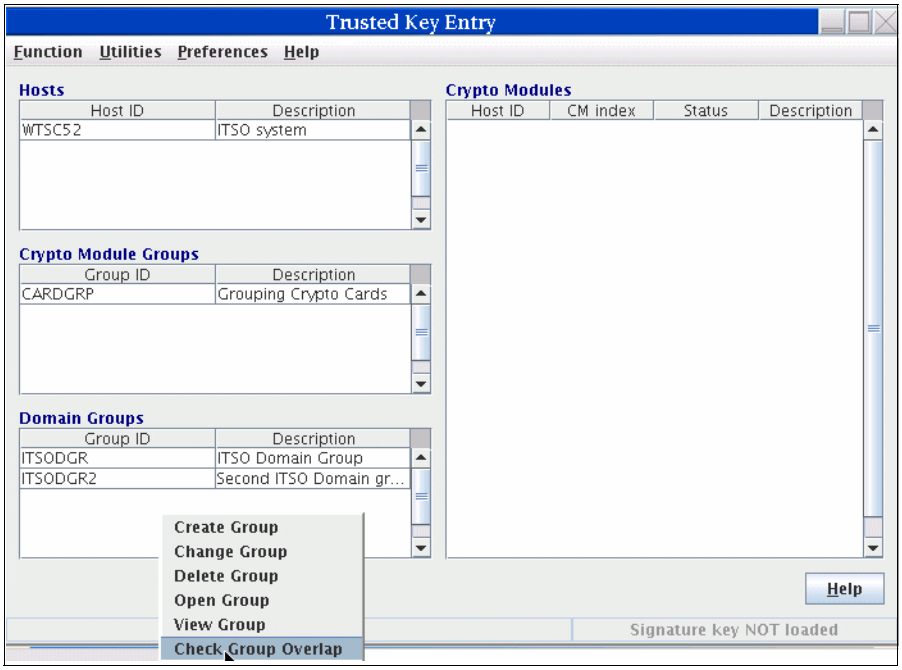


Figure 7-13 Compare the domain groups for overlapping definitions

If overlaps are found, the verification will produce a report window that will show which domains are defined in more than one domain group (Figure 7-14).

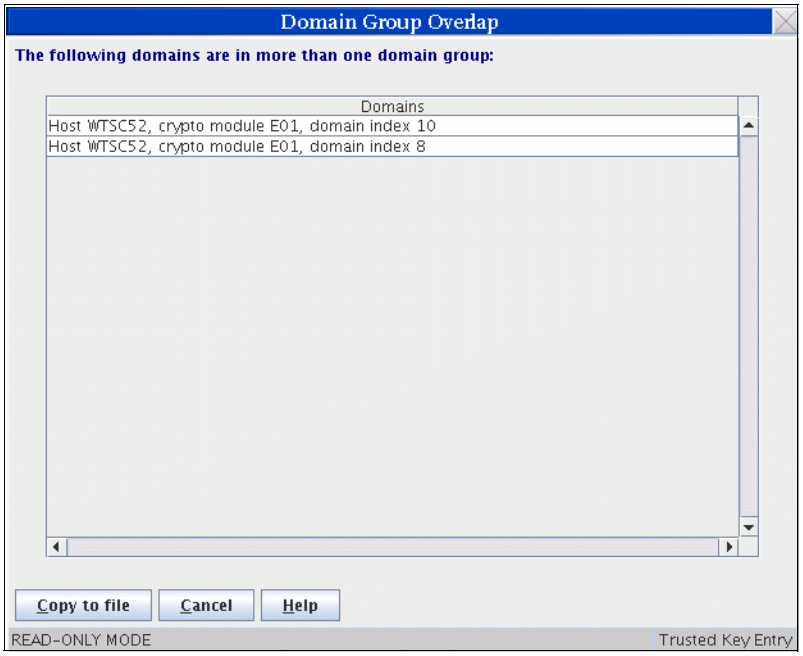


Figure 7-14 Report on the domain group compare

To see overlap details, double-click the overlap report line and a new window indicates which domain groups in which domains are defined more than once (Figure 7-15).

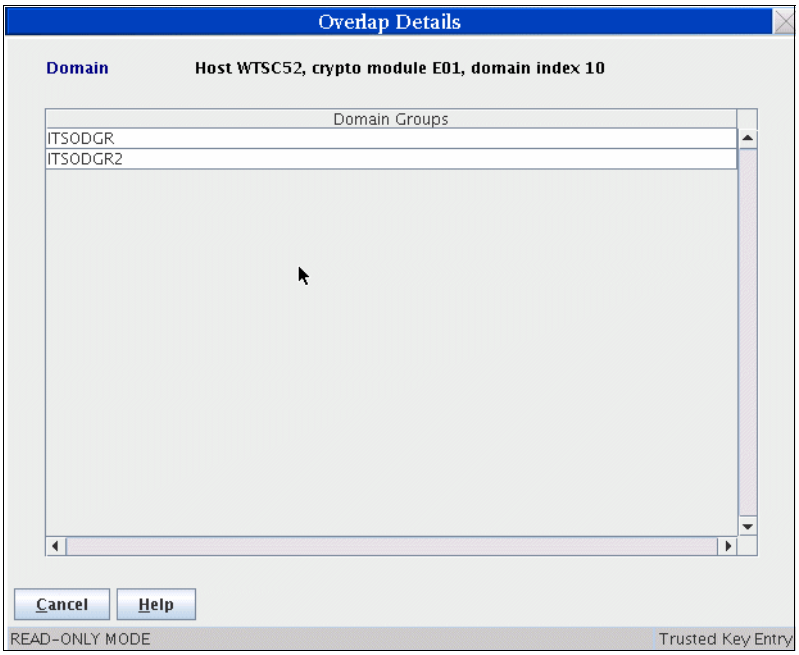


Figure 7-15 Same domain in two different domain groups

Important: If both Crypto module groups and domain groups are defined, there is no tool to verify overlapping definitions. Therefore, we suggest using only one grouping method.

To open a domain group, right-click the desired group and select **Open Group** (Figure 7-16).

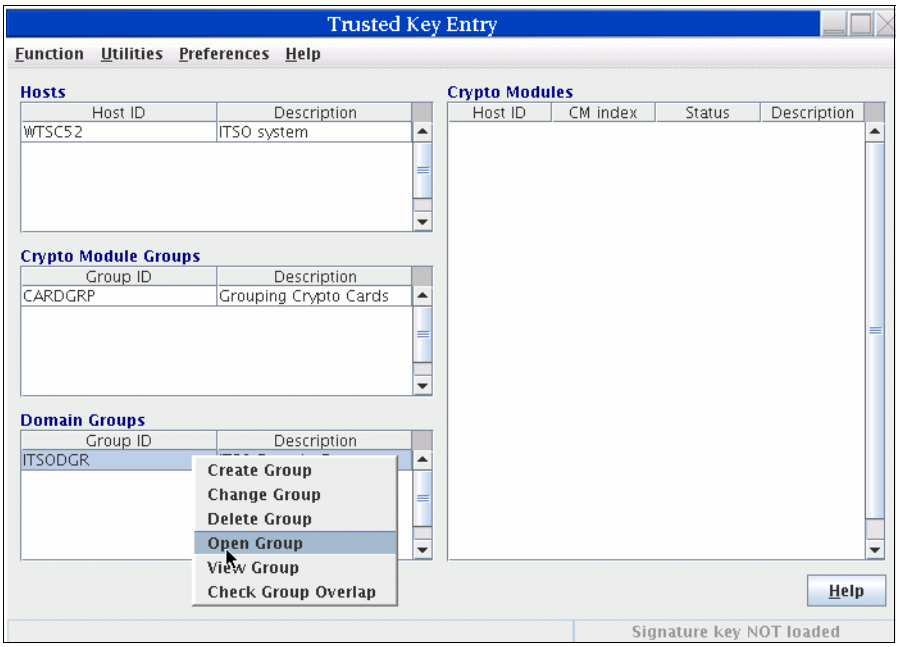


Figure 7-16 Open domain group connection

You might be required to log on to the host by giving RACF or an equivalent user ID and password if the host connections related to the domain group are not opened earlier.

The next window shows the Domain Group Administration window, which is similar to the Crypto Module Administration window, except with the Domain tab the domain selection option is not shown on the right side of the window (Figure 7-17). This view shows the content of the master domain of the domain group.

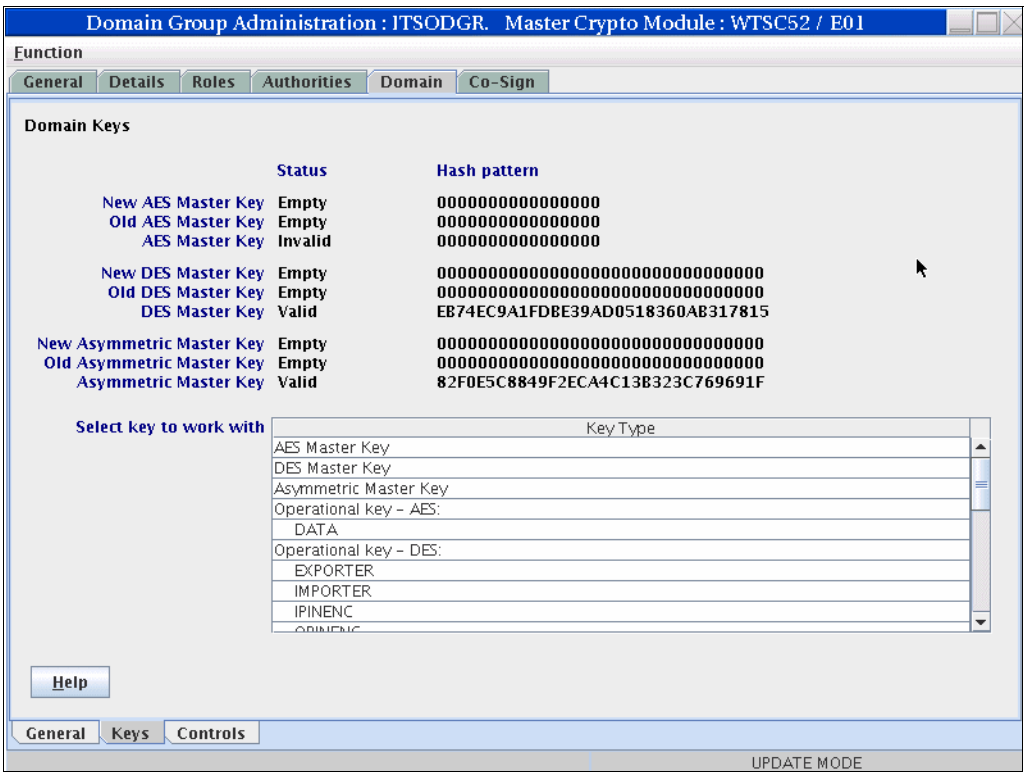


Figure 7-17 Domain Master Key panel

If there is a need to do group compare when using domain grouping, select **Function** and from the list select **Group Compare**. This process compares the content of the master domain against all other domains in the group (Figure 7-18). If there are mismatches in the group, a report is shown that can be used to make corrections.

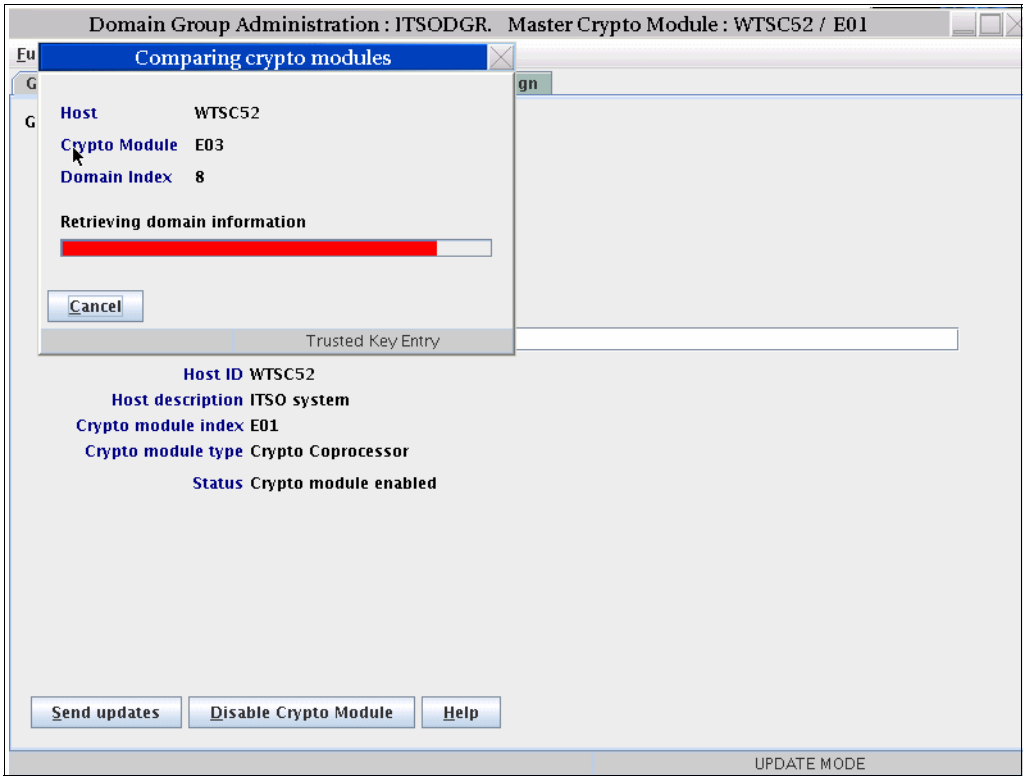


Figure 7-18 Group compare retrieves data on domain level

7.3.3 Support for saving and restoring host configuration data

TKE6.0 has a new capability to copy and store host configuration data. This feature is very useful when a Host Crypto card is replaced or a new Crypto card is added to the host. In addition, when a new host is added to the TKE Crypto Zone, this functionality will help to get faster wanted definitions in place.

Using this functionality, you are able to copy the role definitions and authority settings, such as signature keys and authority and role relationships, and store this information in a file in the TKE and use it later. Also, domain control settings such as ISPF Service settings will be copied and stored. This process makes the base definition setup easier, faster, and less error prone. The storing process will store the exact information saved in the file, and it might not be exactly what is required, but this migration can be used to get the base definitions in place.

Note: The migration process will not copy and restore master key values. Master keys must be entered using existing processes after the roles and authorities have been restored or defined.

Next we go through selected cases showing how to use the Cryptographic card configuration wizard.

To start the migration wizard, click **Migrate IBM Cryptographic Adapter Configuration Data** on the TKE Console under the Trusted Key Entry workspace view (Figure 7-19).

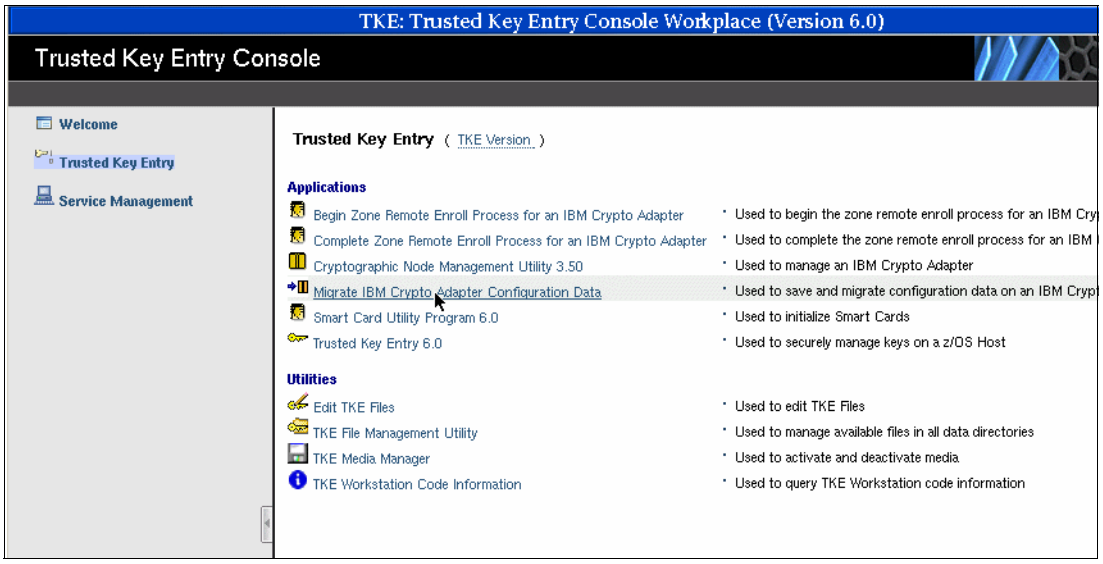


Figure 7-19 Start migration of the host cyrpto adapter configuration task

A welcome window displays first to show what can be done with this Crypto adapter configuration migration wizard. To continue the process, click **Next** (Figure 7-20).

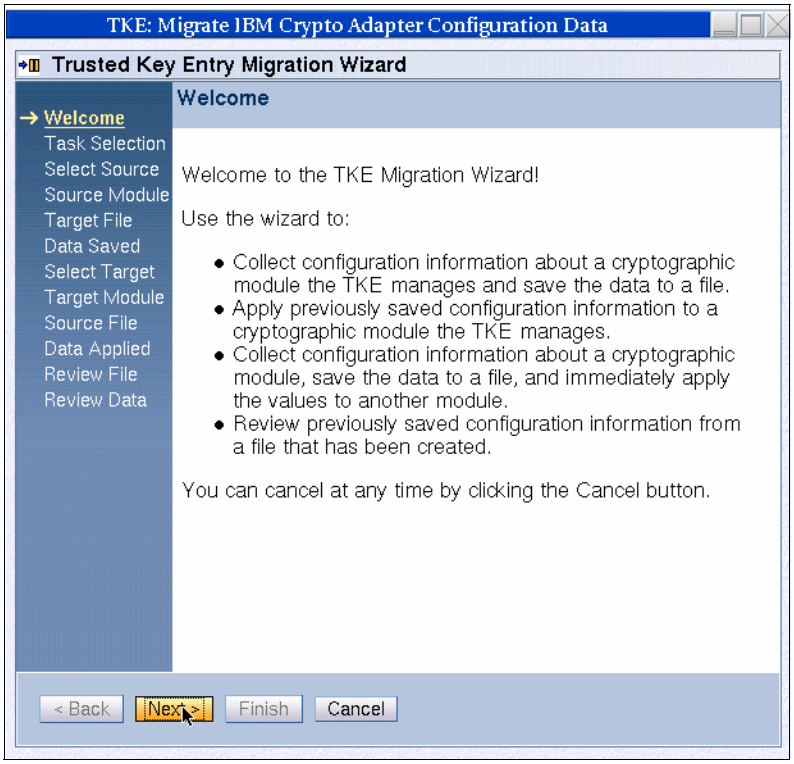


Figure 7-20 Migration configuration welcome panel view

The task requires user authentication to the TKE workstation with the correct privilege. This can be done using the passphrase or the smart card authentication method. We used the passphrase method (Figure 7-21).

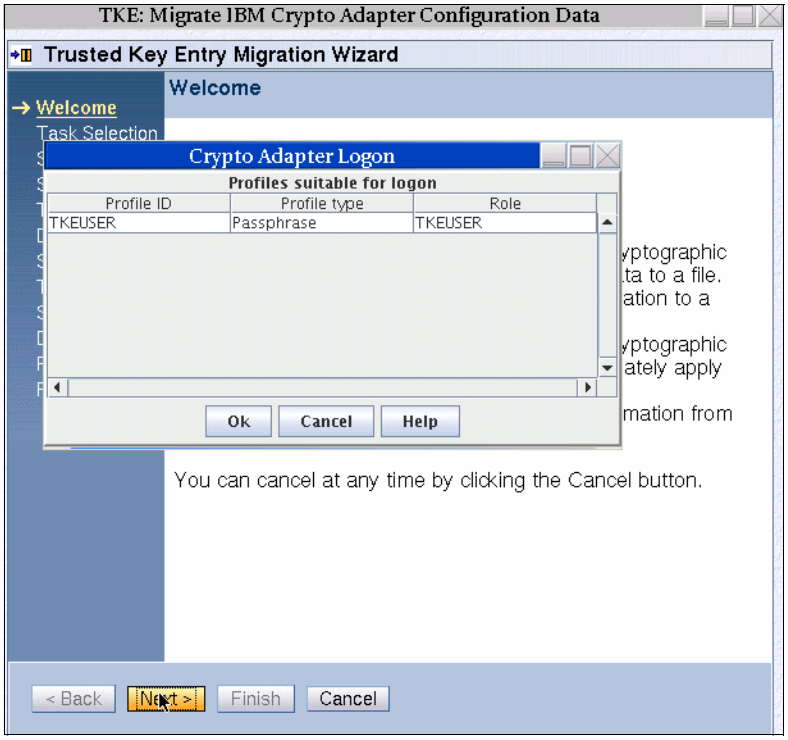


Figure 7-21 Authenticate to the migration utility

After authentication, click **Next** to continue to the Task Selection, where we chose to collect Crypto adapter configuration data, which will then be applied to another Crypto adapter (Figure 7-22). Click **Next** to confirm the task selection.

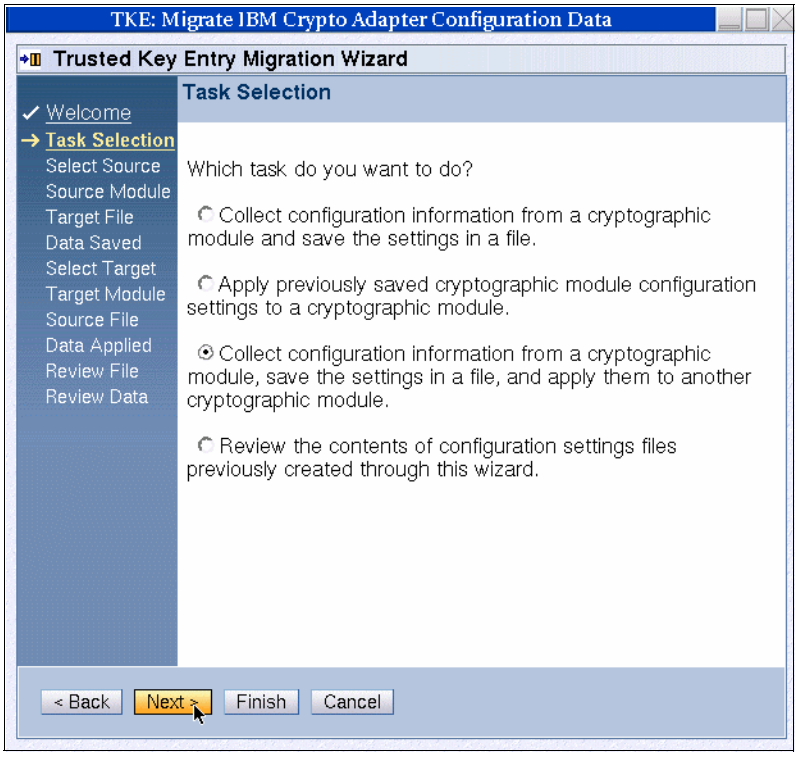


Figure 7-22 Collect configuration settings and save them into a file to apply to a target

For the next step, the source for the Crypto adapter configuration data must be selected. The options here are gathered from the Hosts window or from the Crypto Module Groups window that is defined on the Trusted Key Entry application main page.

If a host is selected (Figure 7-23), it will allow you to choose which of the Crypto adapters will be used as the source to collect Crypto adapter configuration data. If a Crypto Module group is selected to be the source, then the data is collected from the master module of the crypto module group.

Note: LPAR Cryptographic Control Domain settings indicate to which domains access can be established. These settings must be equal on the source and target or migration will not be allowed.

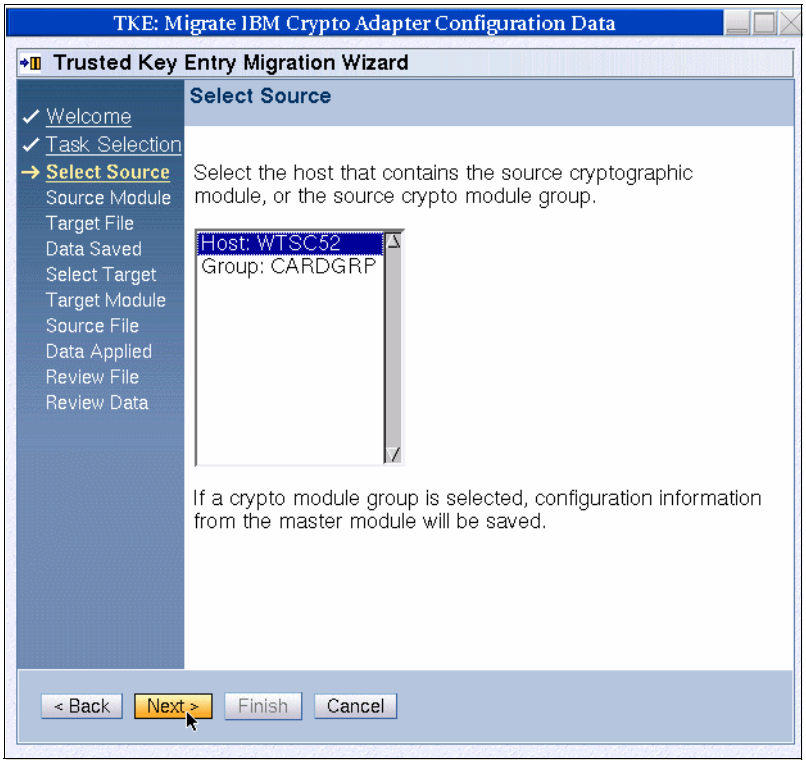


Figure 7-23 Source cryptographic card in the host

To get access to the source crypto adapter configuration, a host logon will be needed by giving suitable RACF or equivalent product a user ID and related password (Figure 7-24). TKE Listener must be running to make the connection successful.

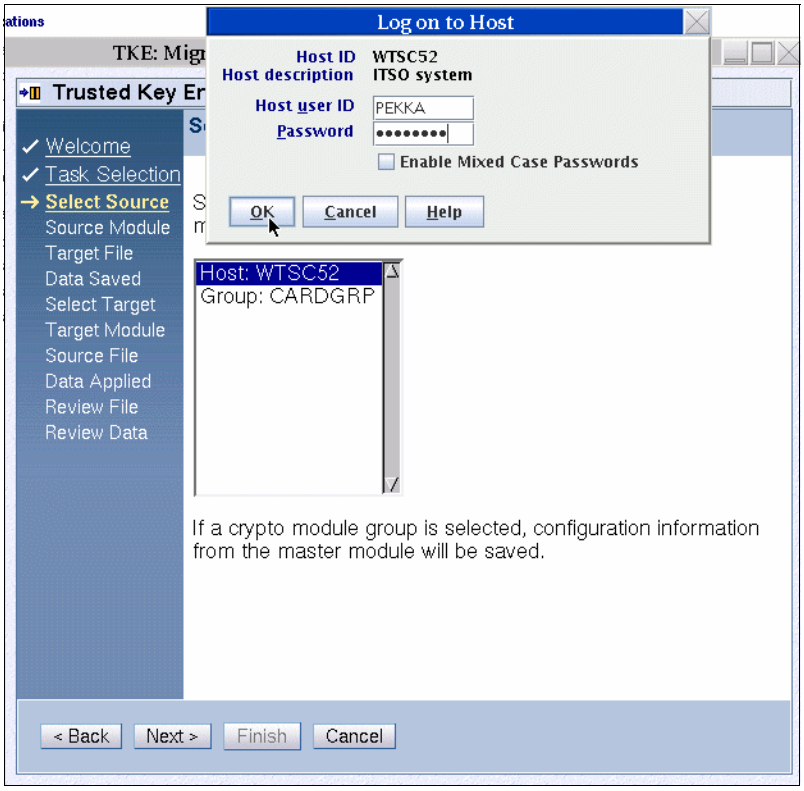


Figure 7-24 Open host connection

We selected crypto adapter 01 to be our source (Figure 7-25).

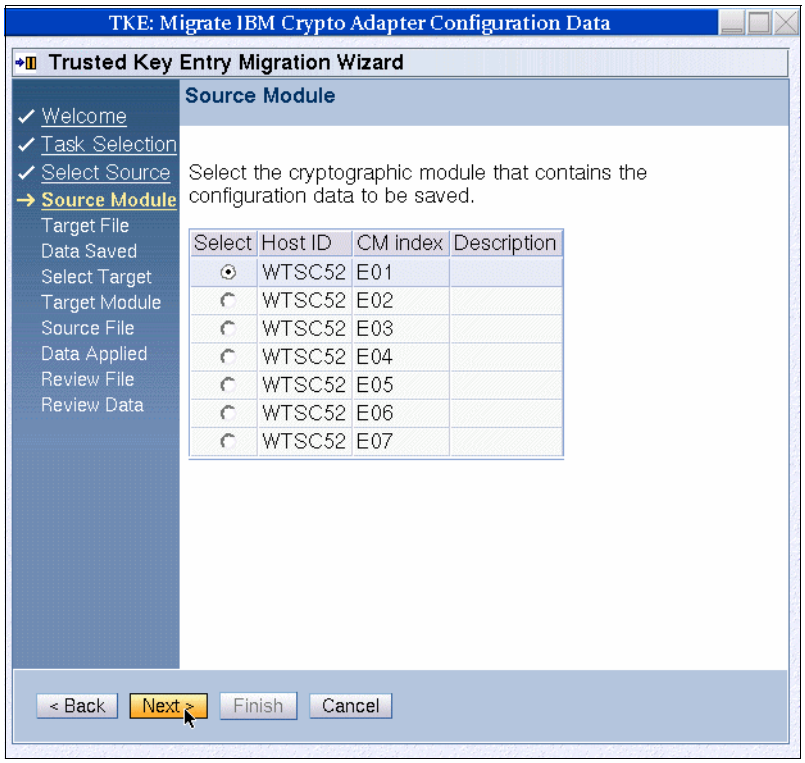


Figure 7-25 Select host source card to collect configuration data

On the next window we select the name of the file where the Crypto adapter configuration data will be saved. This file can be copied to CD/DVD-RAM if necessary.

Clicking **Next** starts the copying of the Crypto adapter configuration data (Figure 7-26).

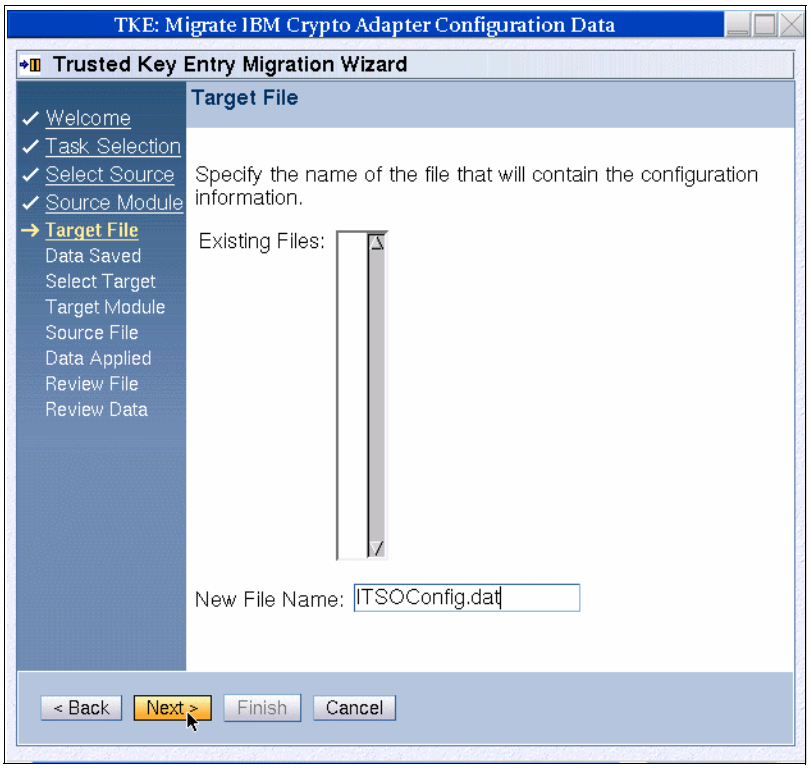


Figure 7-26 File name to store migration configuration data in TKE workstation

When the configuration data is save into the file, the wizard will prompt you to select the target to apply the Crypto adapter configuration data to. If selecting a host, the next selection is the target Crypto adapter (Figure 7-28 on page 202).

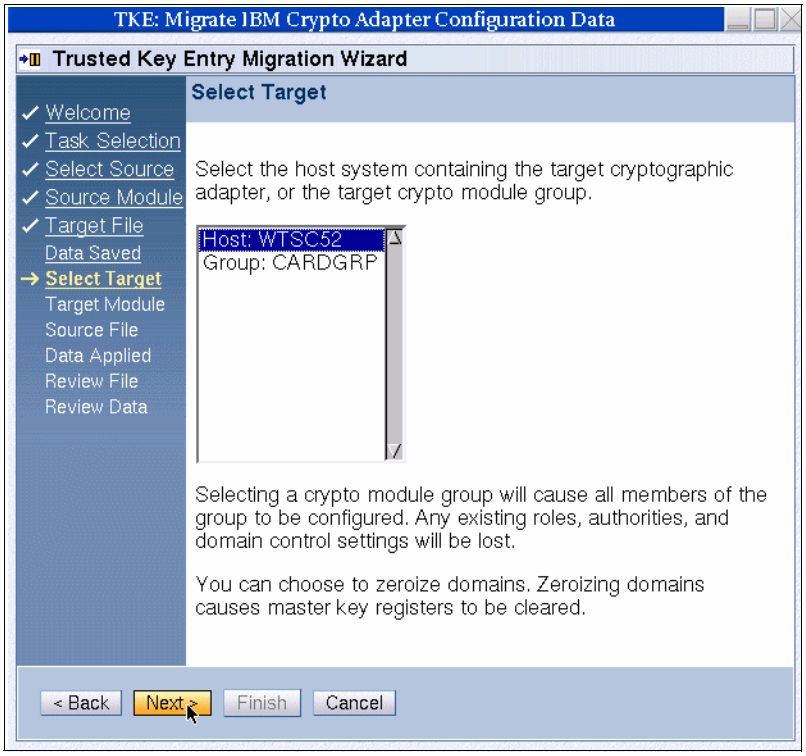


Figure 7-27 Select target host

If the target is a Crypto Module group, then the configuration data will be applied to all Crypto adapters belonging to the group.

On the target Crypto module, all existing roles, authorities, and domain settings will be cleared and the source Crypto adapter values will be used to populate the target Crypto adapter. The process will use temporary roles and authority index 99 for the copying process. These temporary settings will be deleted after the copying has been done.

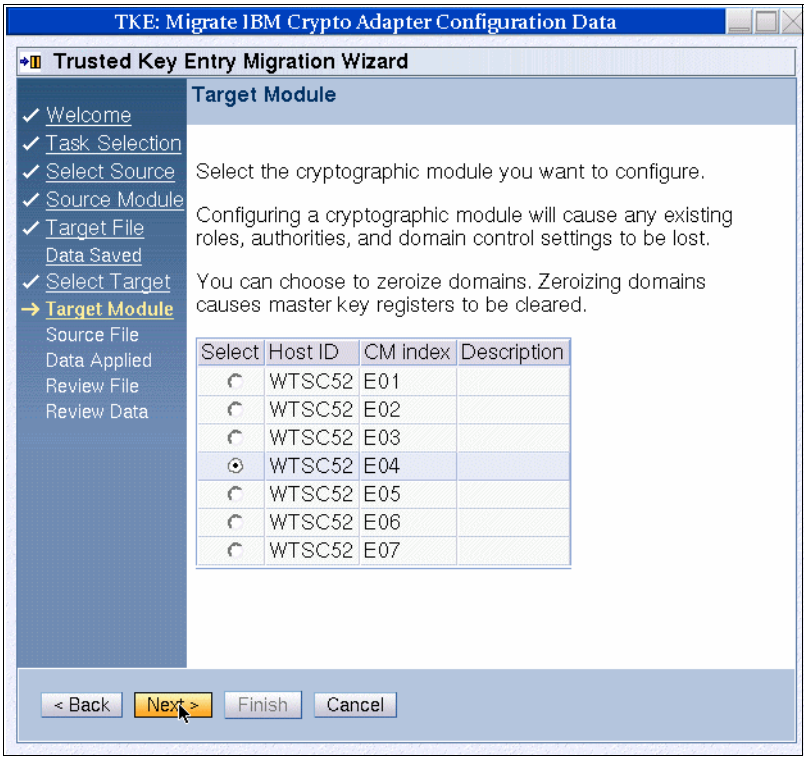


Figure 7-28 The target cryptographic coprocessor

Before the copying process starts, a verification will be done to confirm that the target crypto adapter master key values should be zeroized. This zeroing will happen only if LPAR Cryptographic control domain settings allow the access (Figure 7-29).

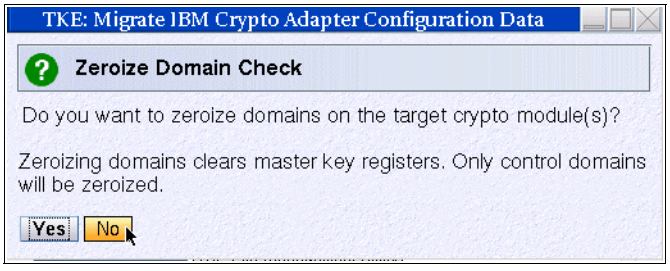


Figure 7-29 Will the target coprocessor domain Master Keys zeroized?

If the domain zeroize is selected and cannot be done for all domains, a warning message window displays and shows which domains cannot be accessed (Figure 7-30).

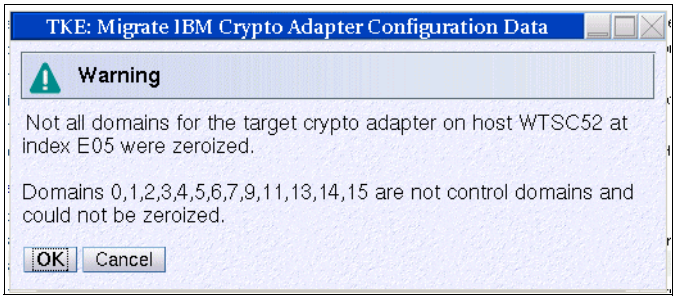


Figure 7-30 Zeroize will be valid based on the control domain settings

To be able to apply the configuration data, a signature key must be loaded. If a new Crypto adapter is the target, then the Default signature key can be used (Figure 7-32).

Figure 7-31 Signature key for the migrate configuration data

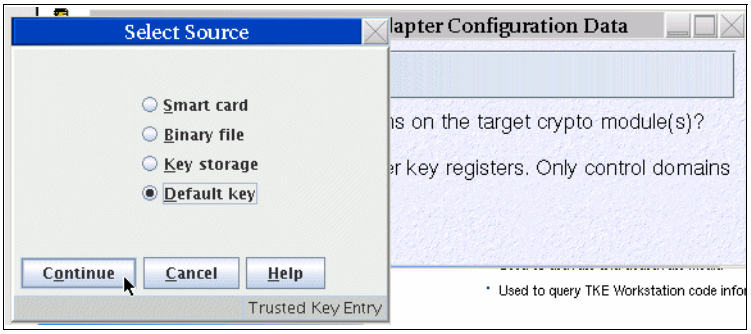


Figure 7-32 Select Default signature key

Then the configuration data will be applied to the target Crypto adapters. This will take some time, so be patient. The apply process will be seen on a separate window (Figure 7-33).

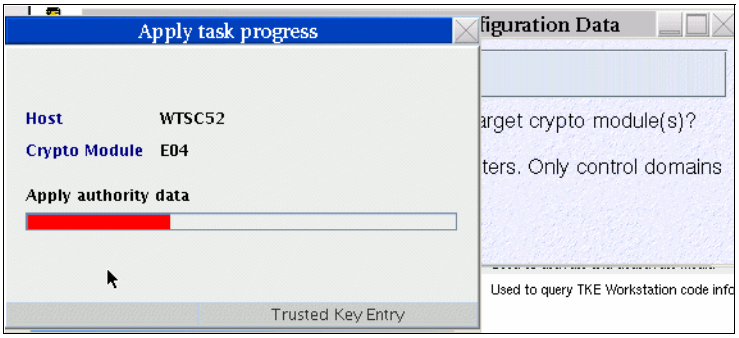


Figure 7-33 Configuration data migration in process

When the Crypto adapter configuration data is applied to the new crypto adapter or a group of adapters, then a new task can be selected or you can finish the wizard (Figure 7-34).

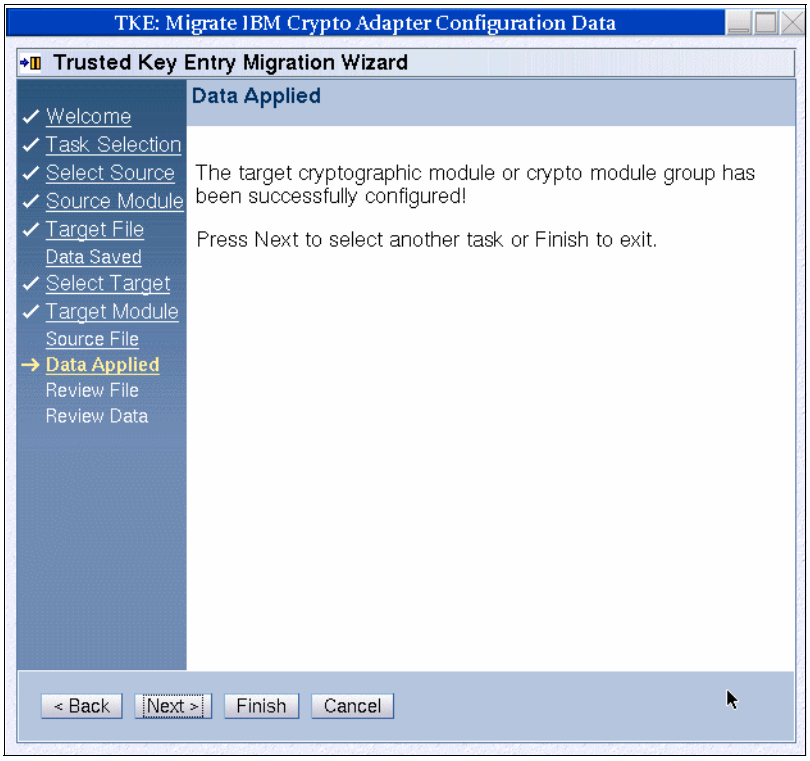


Figure 7-34 Select a new task or finish

To view the content of the Crypto adapter configuration file, we select the appropriate task (Figure 7-35), and then click **Next**.

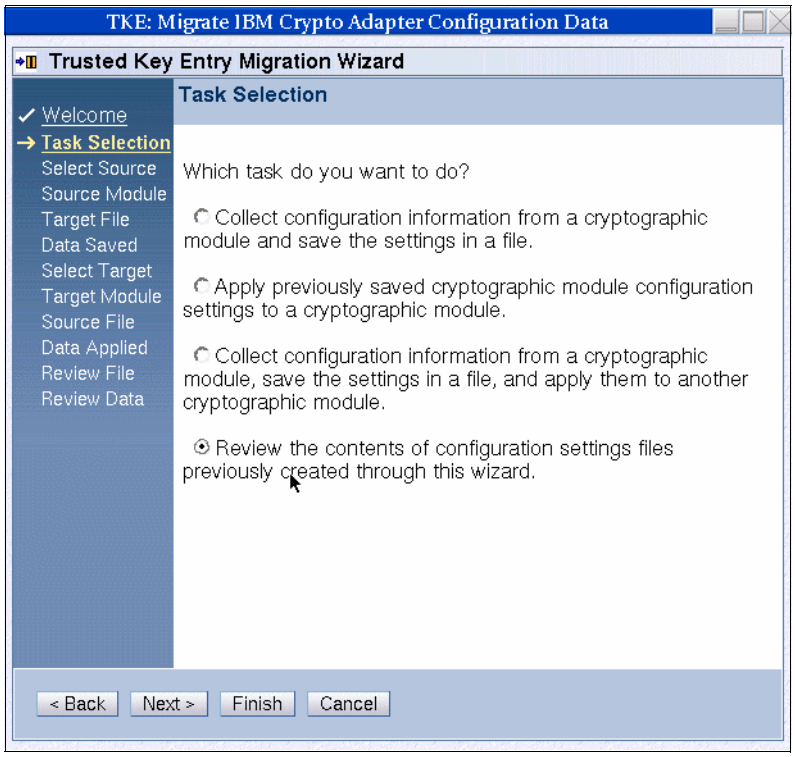


Figure 7-35 Review the collected configuration data

First, select which configuration file to view. Highlight file name and click **Next** (Figure 7-36).

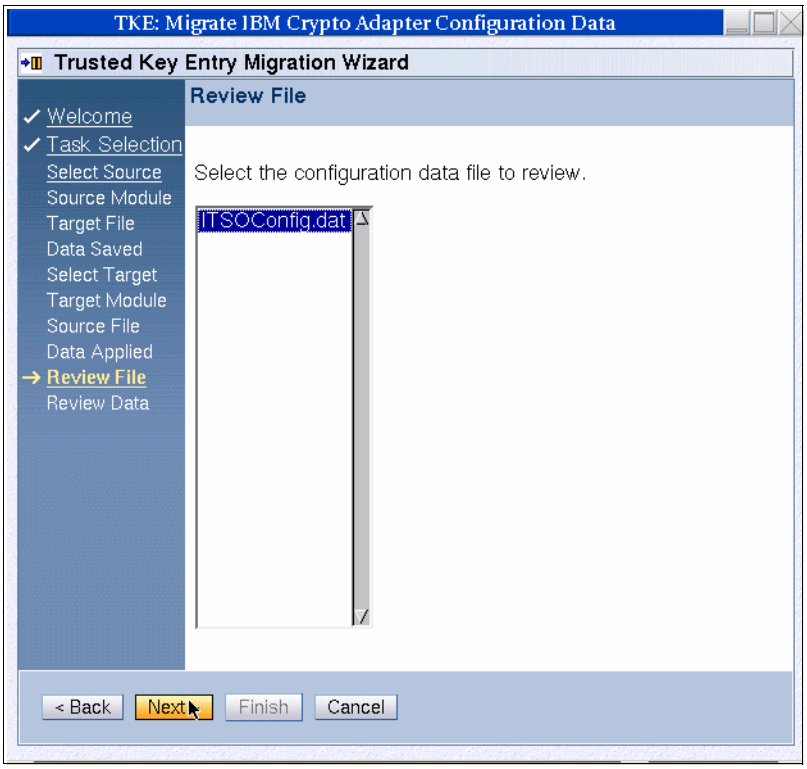


Figure 7-36 Select the correct file for review

There are several tabs: Details, Roles, Authorities, and Domains (Figure 7-37 and Figure 7-38). All information gathered into this file is Public (that is, it contains no secret information, such as master key values).

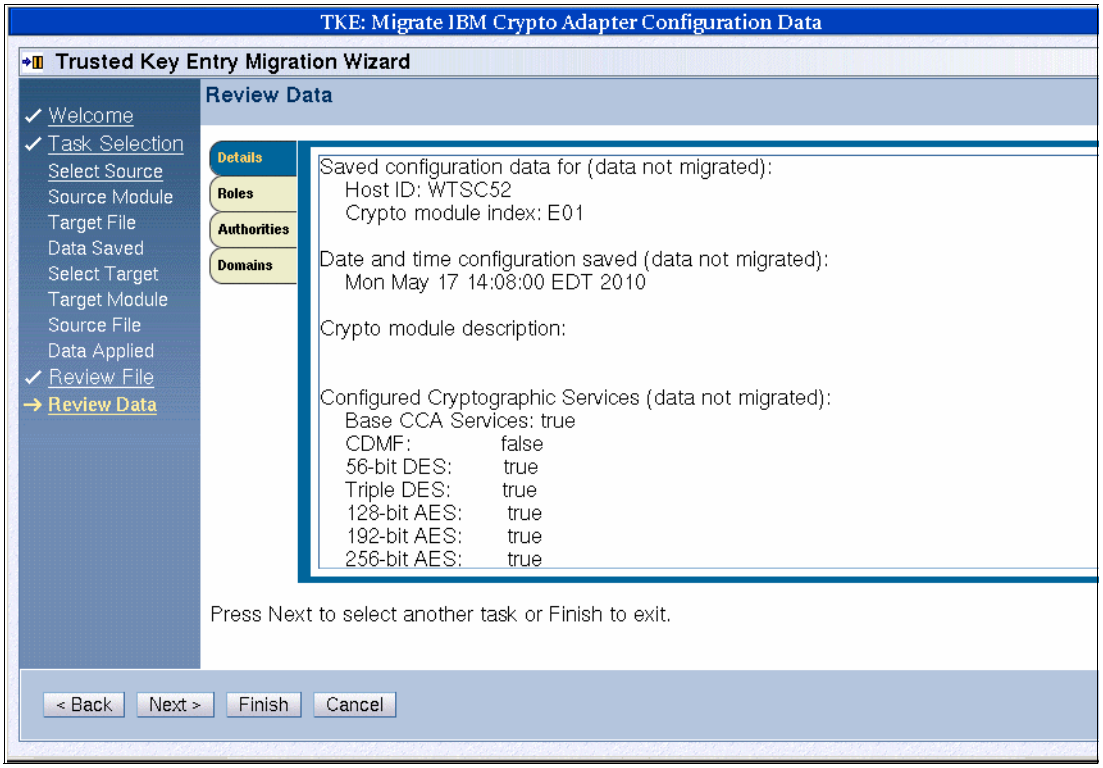


Figure 7-37 Collected configuration data - Details view

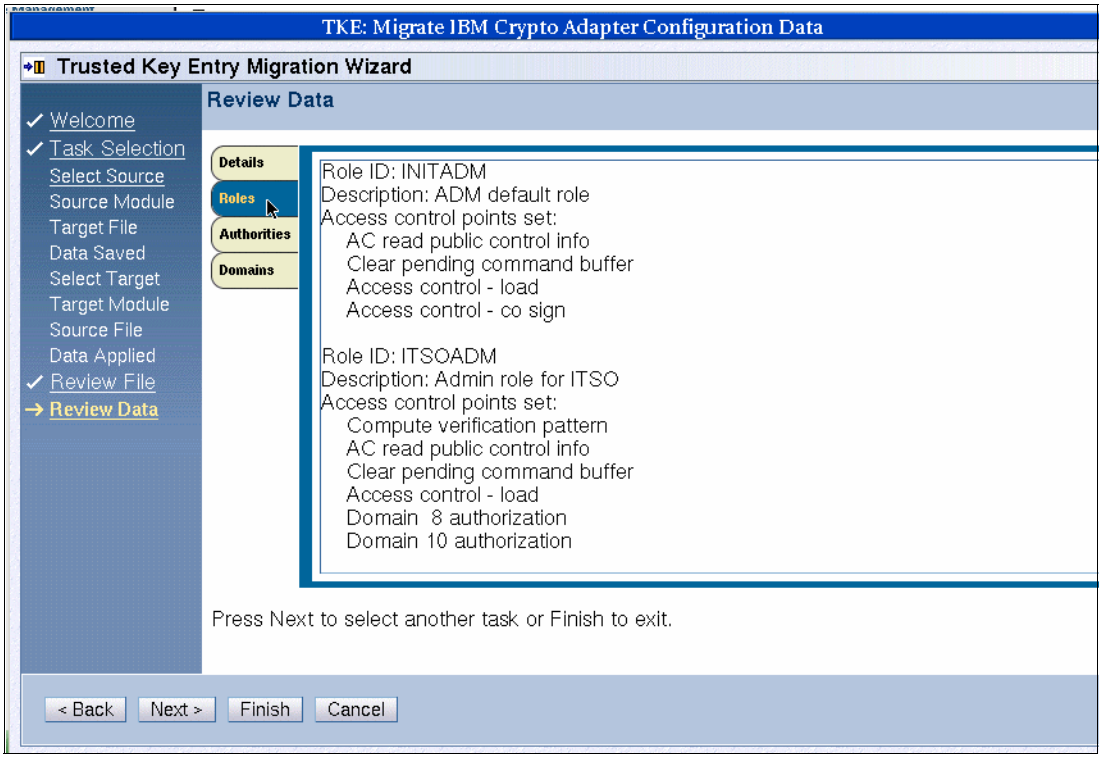


Figure 7-38 Collected configuration data - Roles view

7.3.4 Admin and auditor tasks

On earlier TKE versions (before TKE5.3), some TKE workstation initializations were executed using the so-called temp default role, which was set on the TKE workstation crypto adapter initialization phase. This very powerful role was quite often left to be active on the TKE workstation due to fear that something might go wrong and so cause a system failure.

We now have a new way to make initial TKE settings and other privileged mode processes easier and less error prone. The TKE workstation can be run in three new user ID modes, each providing special privileges. These are:

- ADMIN

  This is used for the TKE initial setup process. This mode is very powerful can be used on almost any purpose on the TKE workstation. It should have limited usage only.

- AUDITOR

  This is used for auditing purposes. This mode has quite limited capabilities but is needed when TKE usage is audited.

- SERVICE

  This is for IBM Service. It is a reserved process and should not be used by the customer.

At power-on, the TKE workstation will start in TKEUSER mode. To be able to log on with the privileged mode, close the main window by clicking the **X** on the upper right corner. A new panel displays (Figure 7-39). Click **Privileged Mode Access**.

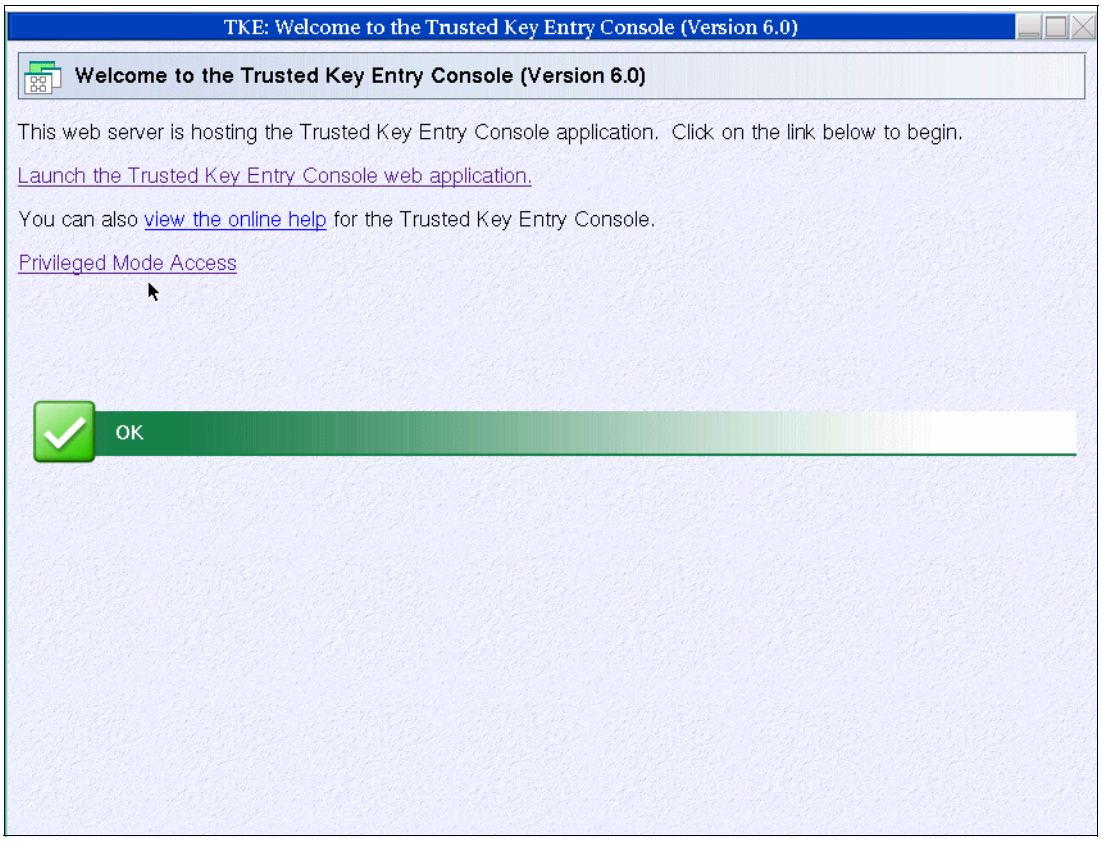


Figure 7-39 Select privileged mode access

ADMIN

A new window displays where the user ID is ADMIN and the password is PASSWORD. The password values are shown in the TKE manual, so they should be changed for better security (Figure 7-45 on page 212).

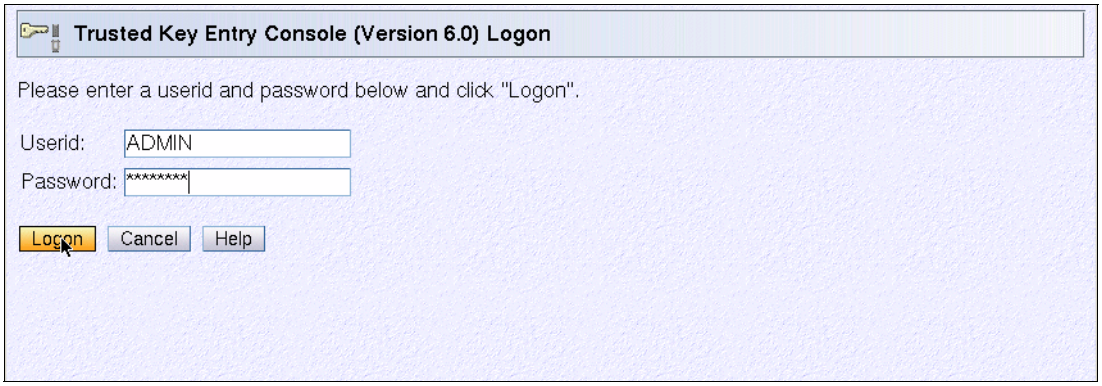


Figure 7-40 Log on as a ADMIN user

After successful logon, the TKE console workspace window opens showing ADMIN-related applications and utilities (Figure 7-41).

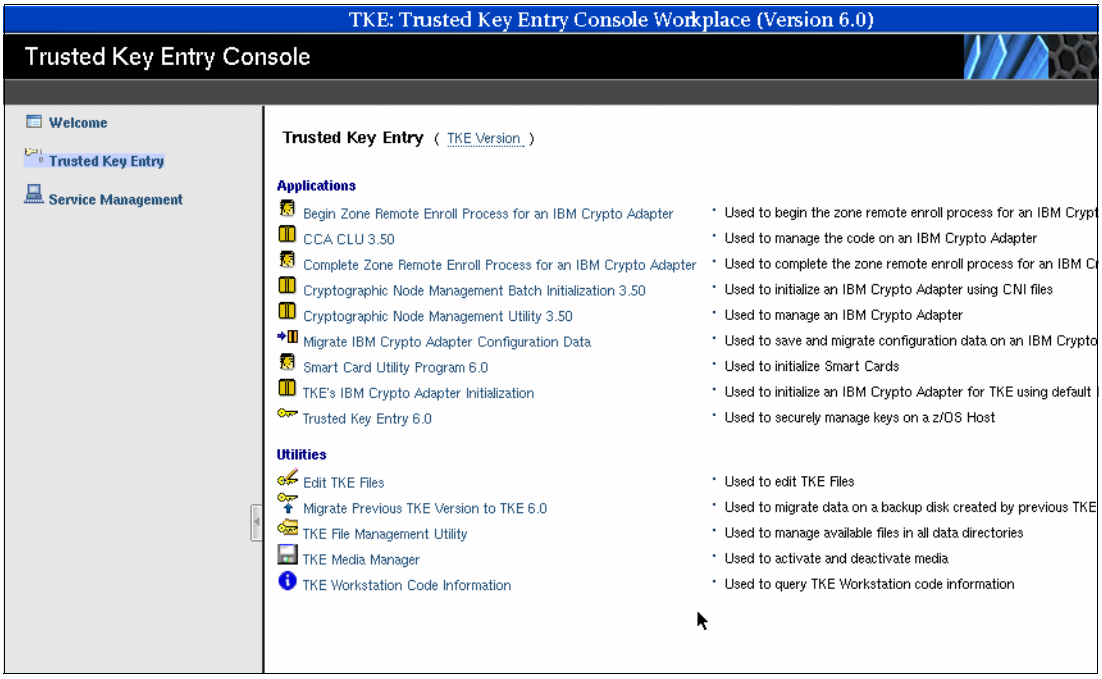


Figure 7-41 Applications and utilities for ADMIN

The TKE code update to use ADMIN instead of the older temp default role will also be seen at the task logons when they are needed. For example, a logon to the Card Node Management (CNM) utility, which is used during the initial setup, will show a logon panel where Default Logon can be selected (Figure 7-42).

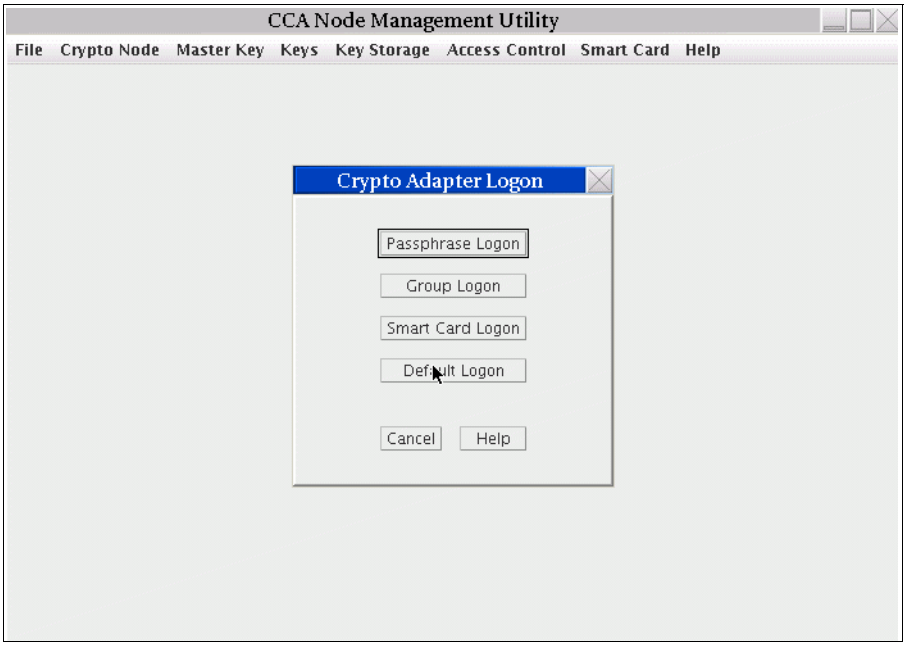


Figure 7-42 Default logon will be used for initial TKE workstation configuration (CNM)

When logging on to the other applications or utilities that require authentication, a logon window (Figure 7-43) appears, where a correct logon is required to start the wanted application or utility. The Crypto adapter logon window will show all profiles that are suitable for the logon.

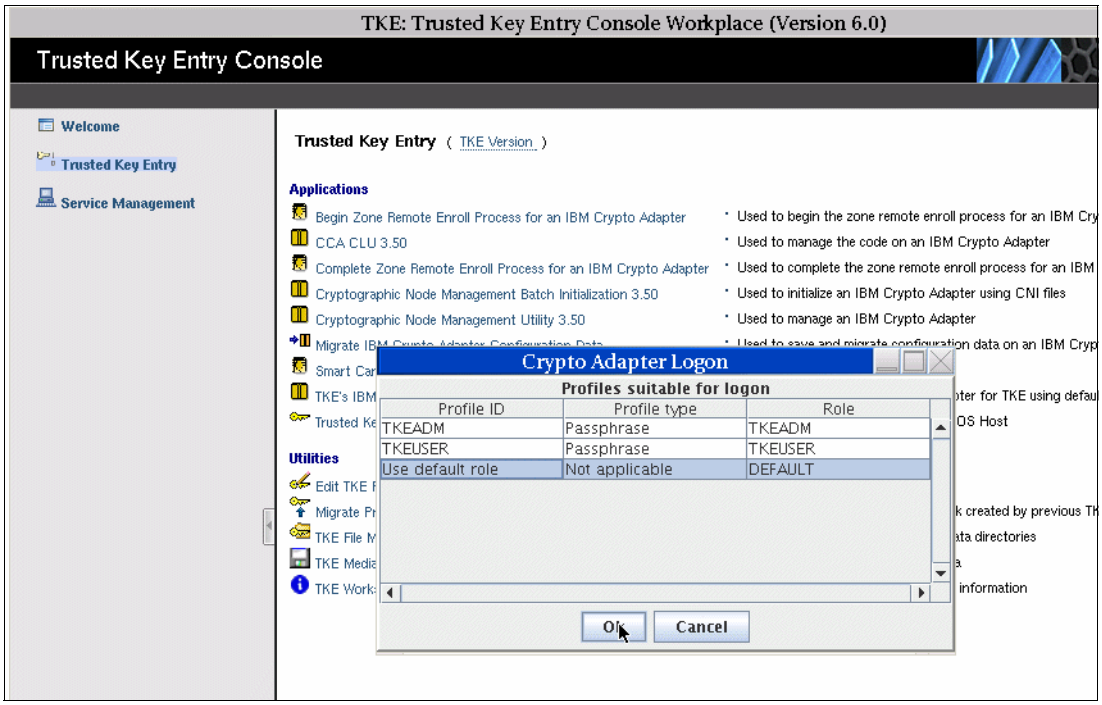


Figure 7-43 Default logon with ADMIN task to configure TKE workstation

Also on the Service Management workspace (Figure 7-44), there are tasks that will be seen only under admin, such as customize network settings for the TKE connections to the hosts.

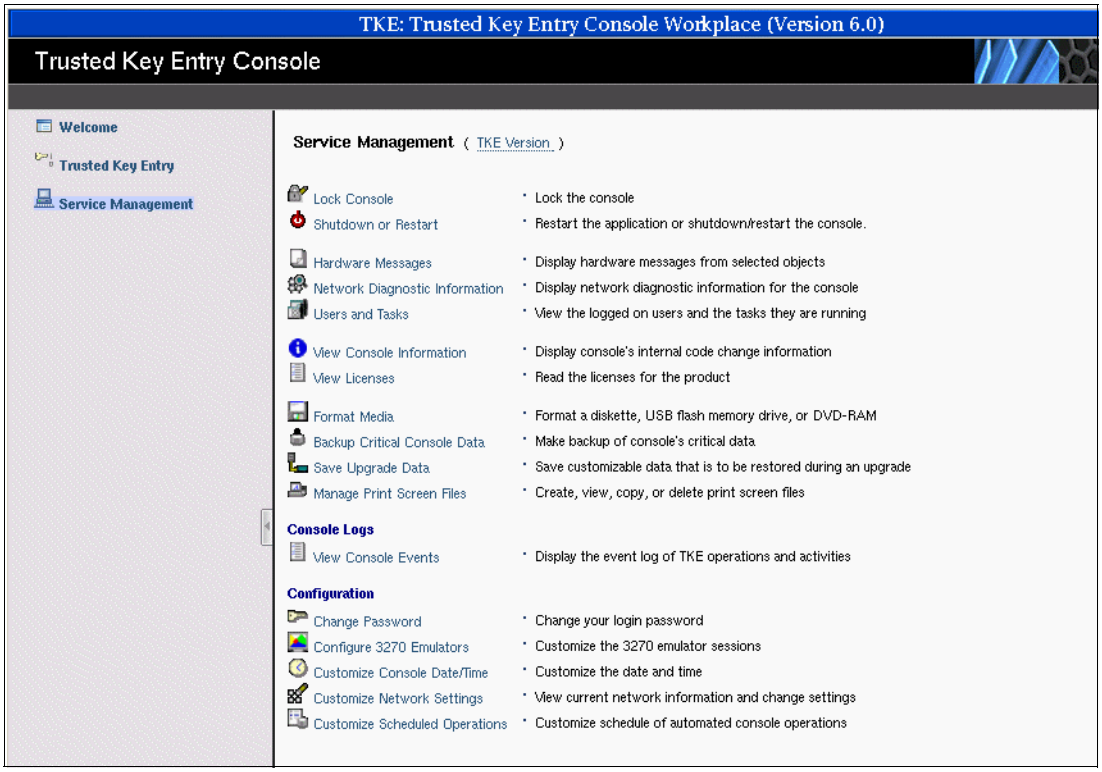


Figure 7-44 Service management for ADMIN

With a powerful tool like ADMIN it is important to change the default password. We also strongly suggest changing the password of the AUDITOR user ID. For the password change there is a task under service management (Figure 7-45).

Note: There is no process to make the usage of ADMIN dual custody. We suggest dividing the password into two parts for two different persons if applicable.

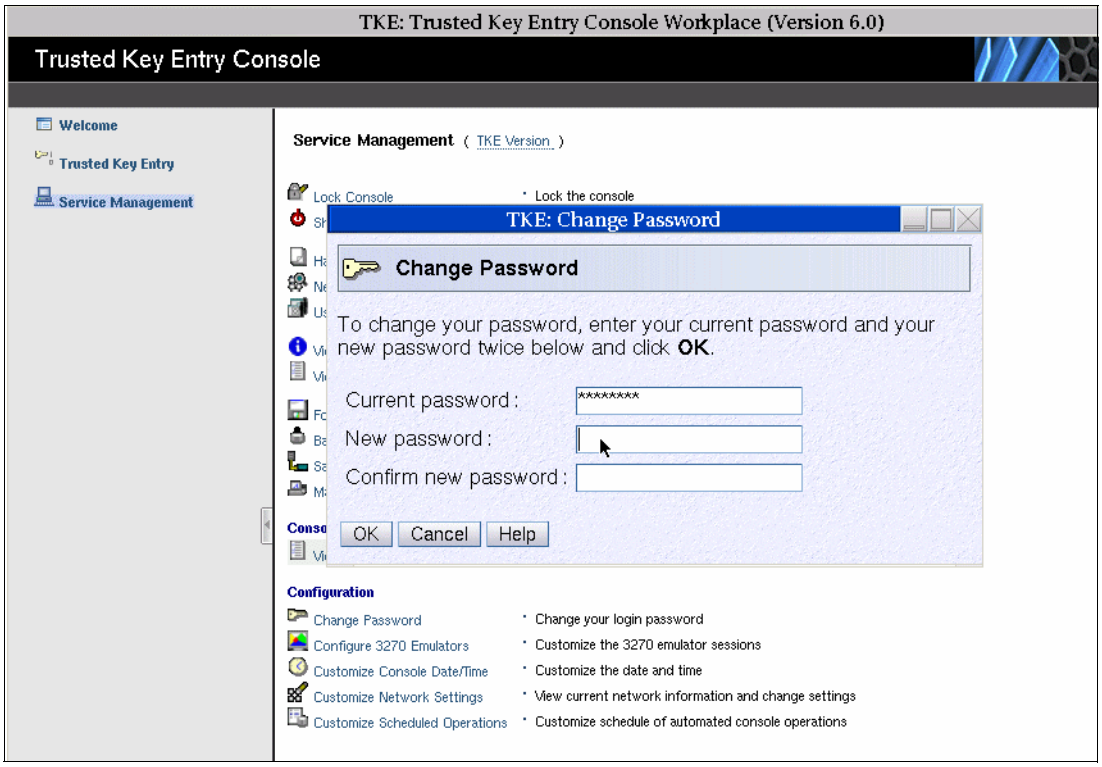


Figure 7-45 Change the factory-set default passwords

AUDITOR

To use the TKE audit capability, you will need to make a privileged mode access logon using AUDITOR as user ID and PASSWORD as a password if not changed yet.

The Trusted Key Entry workspace shows tasks available for audit purposes. The first step is to configure auditing for your company needs. Click **TKE Audit Configure Utility** on the workspace (Figure 7-46) and a new window will open showing the current TKE audit settings.

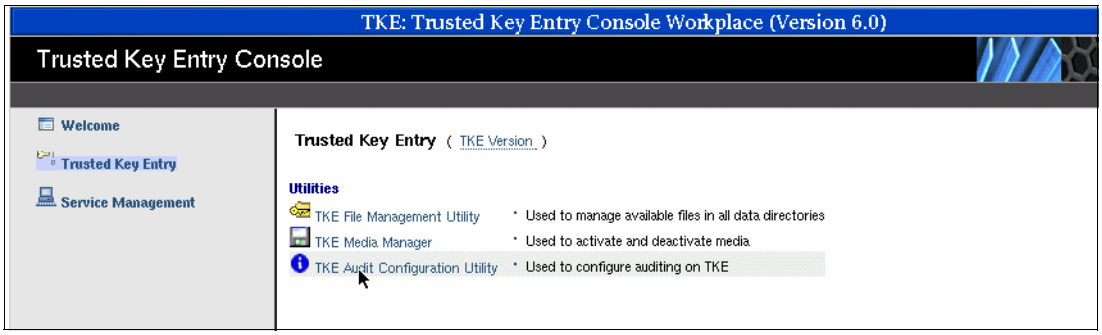


Figure 7-46 Configure TKE audit settings

TKE comes from the factory with all audit points turned on as a default. All successes and failures will cause audit record creation on the TKE hard disk. The Audit Configuration Utility panel (Figure 7-47) will show the main audit points.

In the audit window the Stop Auditing button turns auditing off completely.

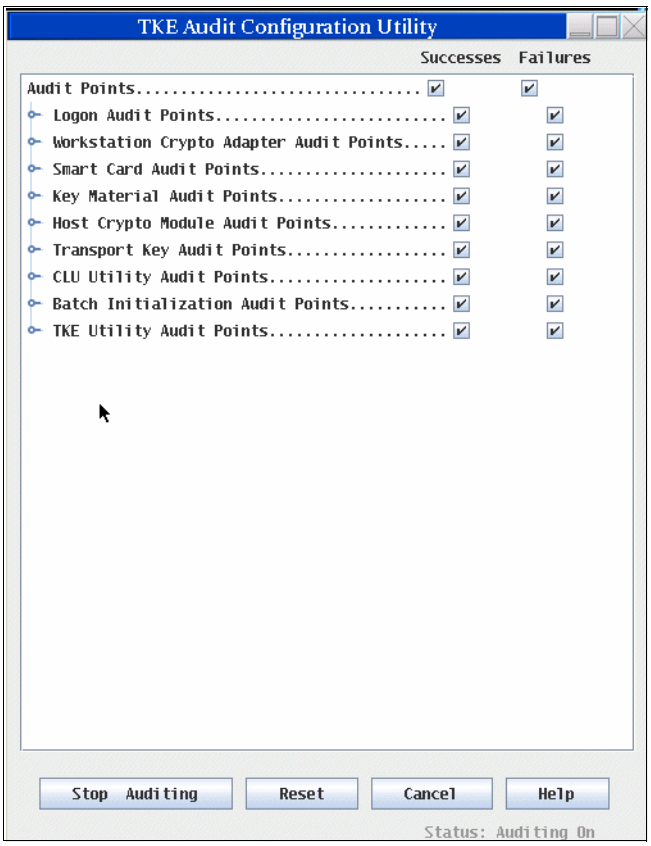


Figure 7-47 As a factory default all is selected

The main audit points can be expanded by clicking the small marker to the left of the audit point to show more detailed audit points on each of them. If all audit points will not fit into the window, you can drag the bar on the right to see other audit points.

Note: Having all audits points active will create a large amount of audit records, and we suggest that you modify the audit configuration to get only those audit records required by the local TKE usage processes and policy.

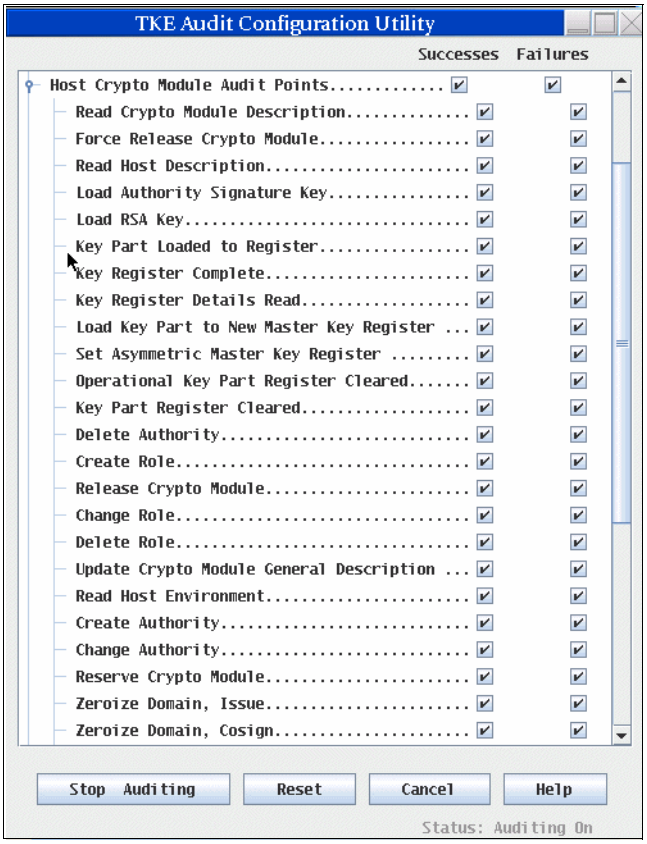


Figure 7-48 Expand to configure detail audit needs

In Figure 7-48 on page 214, the Host Crypto Module audit points are expanded to show more detailed audit points.

Note that ICSF will also record certain events related to the TKE activity. The events will be recorded on the SMF record 82 subtype 16 every time that TKE issues a command request to, or receives a reply response from, the Host Crypto module. This SMF record does not give as much detailed information as the TKE audit log.

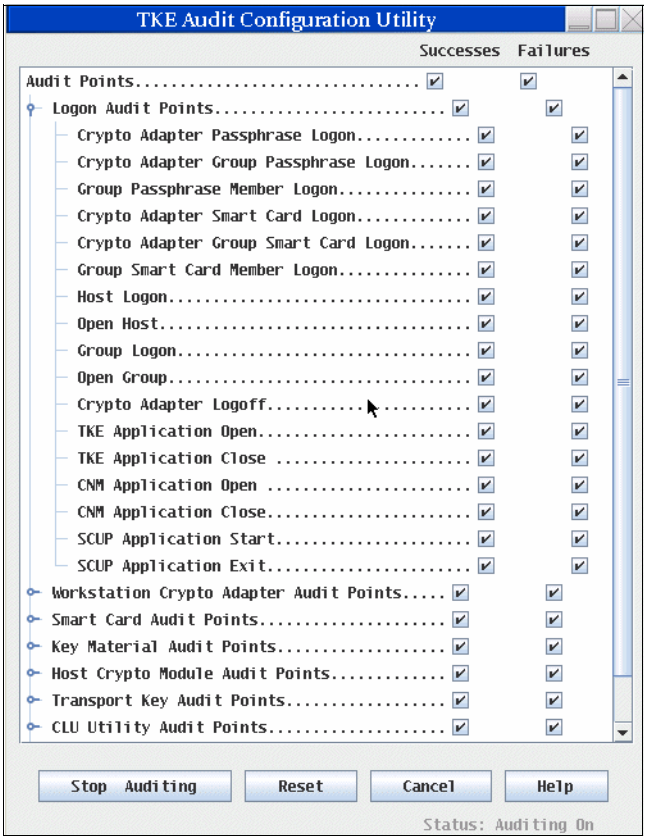


Figure 7-49 Audit configuration detail possibilities

The TKE logon audit point is expanded in Figure 7-49 on page 215 to show the capabilities for audit TKE internal processing.

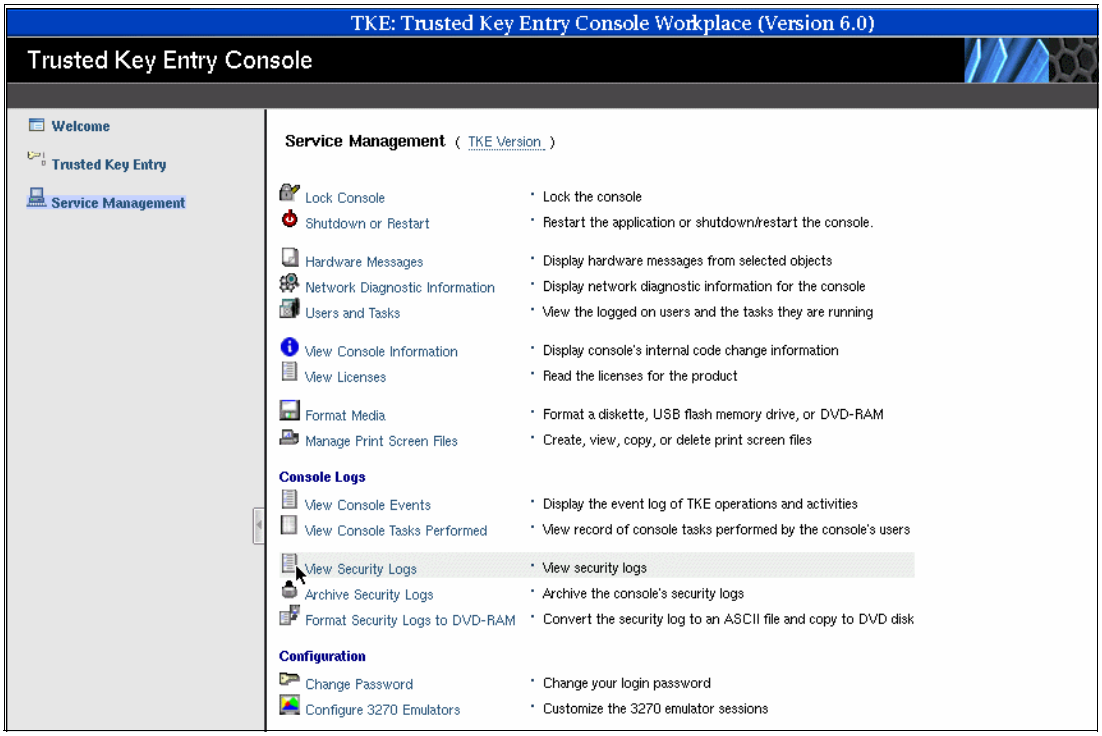


Figure 7-50 How to view audit security logs

To review the TKE audit logs, select the **Service Management** workspace and then **View Security Logs** (Figure 7-50). This option is available when privileged mode access has been logged on with AUDITOR.

Note: The security log has a size of 30 MB, and when the log is 100% full, the oldest third part is deleted. When the security log is 75% full, a warning message is displayed.

Figure 7-51 shows the audit record view, which provides information about the date and time information and what has been done, and by whom (= TKEUSER profile).

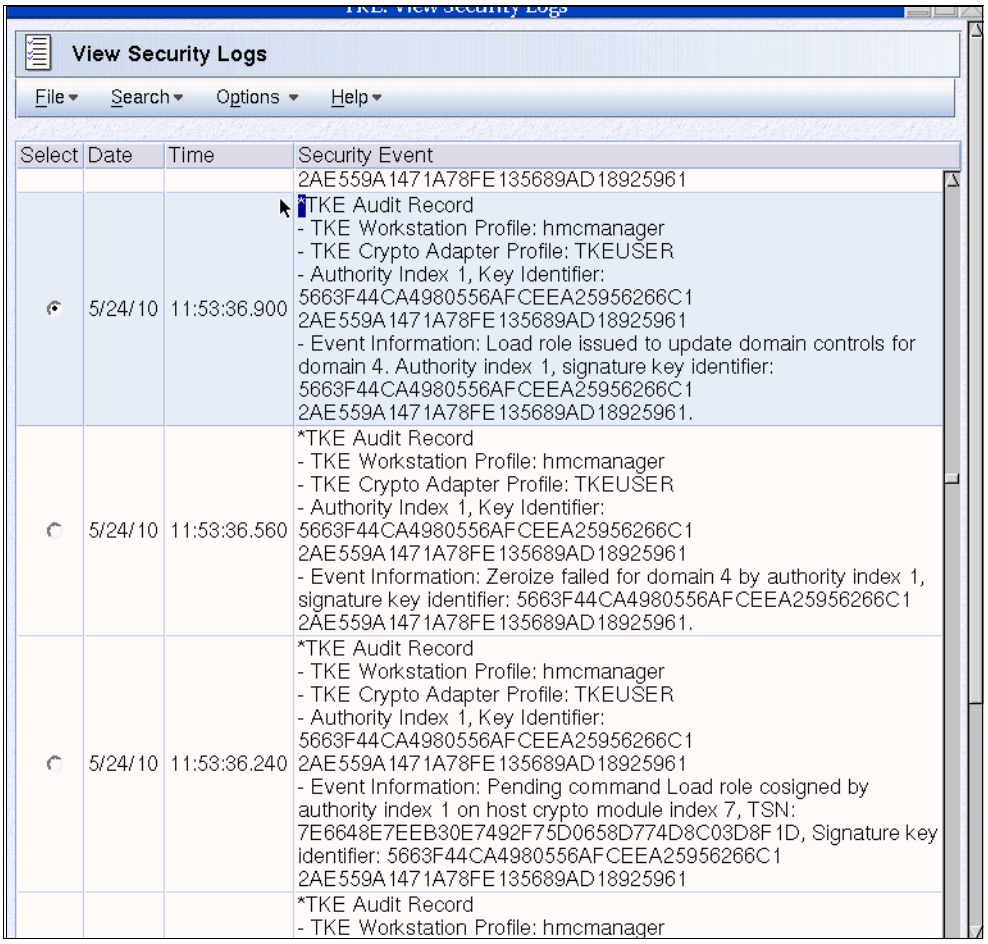


Figure 7-51 Audit record view

If there is an asterisk (*) next to the audit record header (see the cursor location in Figure 7-51), there are more details available for the audit record. Select the corresponding TKE audit record and click **Details** at the bottom of the window (not shown in Figure 7-51). A new window displays that shows detailed information (Figure 7-52).

Figure 7-52 Audit record detail view

The TKE audit records are shown as pages that contain 1000 records. To view these audit records, there are two bars on the right side of the window that can be used to view the log (see the cursor position in Figure 7-53). The pages also can be navigated using the Show Earlier Events and Show Later Events buttons at the bottom of the window.

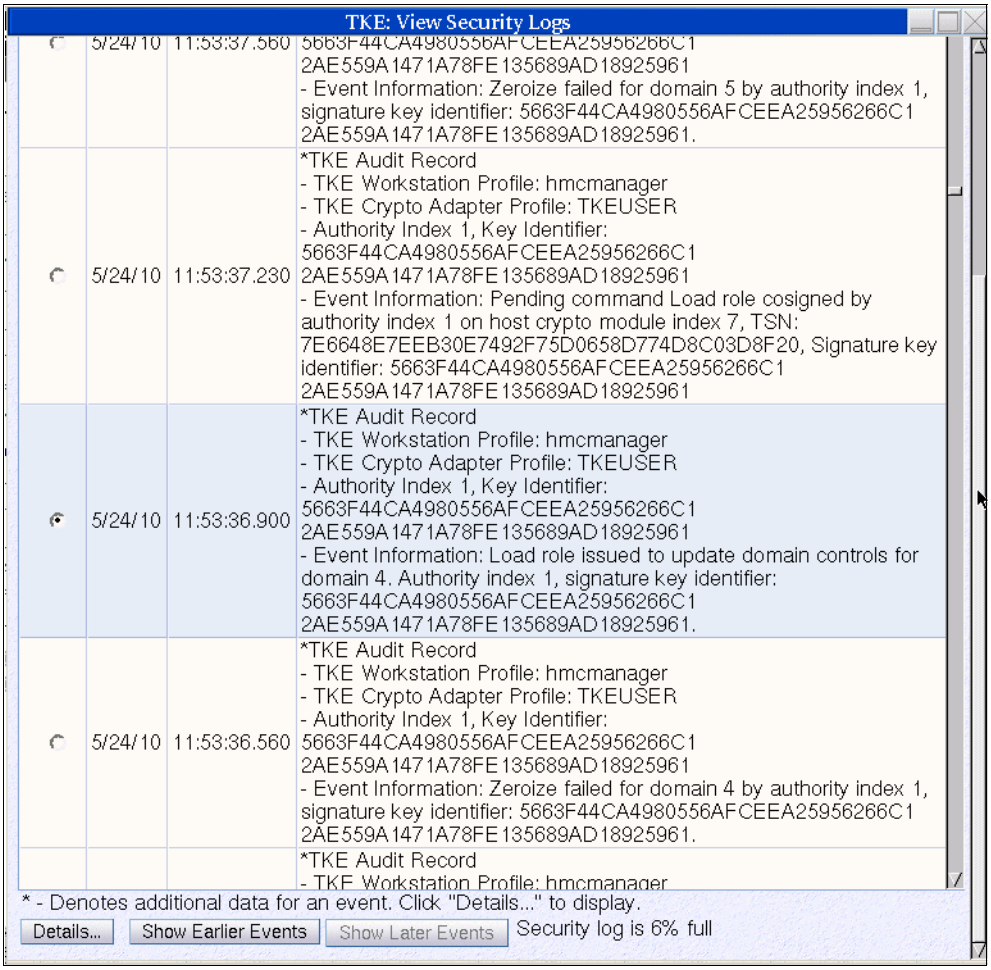


Figure 7-53 How to scroll audit record view

Audit log archiving and formatting security log

Audit log data can be archived to the DVD-RAM. This requires that the DVD-RAM has been formatted using the TKE. The archive process deletes archived logs from the TKE up to 20% of the maximum capacity.

When selecting Format Security Logs to DVD-RAM, the process will copy the content of the security log to DVD-RAM in ASCII file format. The DVD-RAM must be formatted either with no label or with label ACTSECLG.

7.3.5 TKE application changes

In this section we introduce other changes that are implemented in the TKE application. These new features do not significantly change TKE usage, but they are enhancements to usability or security.

Host role and authority changes

The host role window structure has been changed to a new format. The main view shows role access control points in list format, where each of the objects can be expanded by clicking the small marker on the left. The role access control point is selected by ticking the desired check box for the role (Figure 7-54).

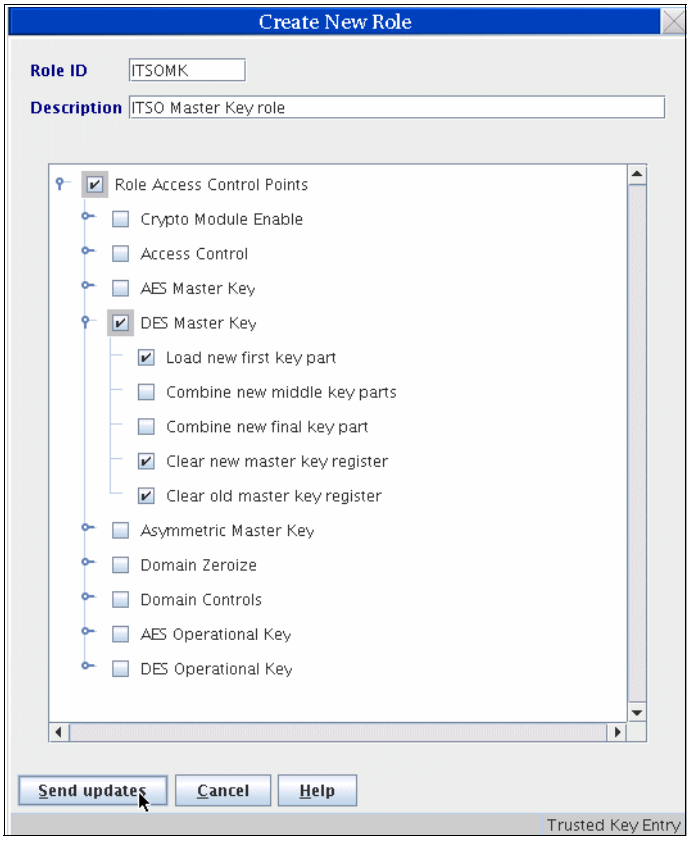


Figure 7-54 The new layout on the role selection window

Changes on the signature key generate

When an authority is created, the first step is to generate the authority signature key, which can be saved in key storage (which has space for one signature key only), in a binary file, or in the TKE Smart card (Figure 7-56).

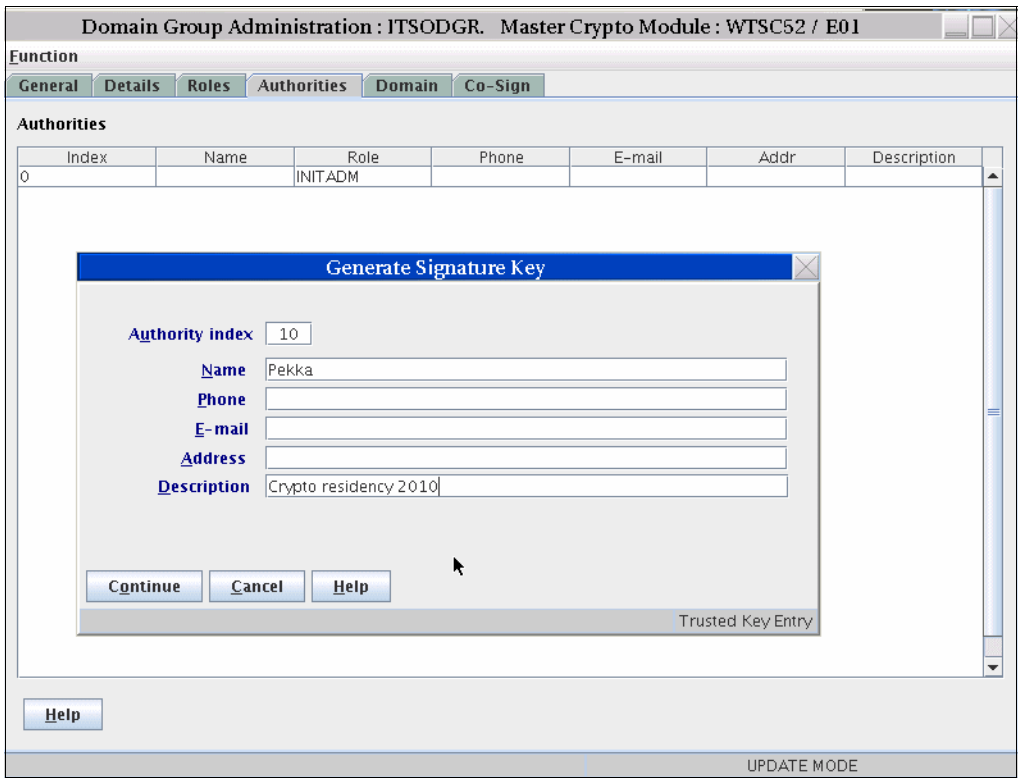


Figure 7-55 Generate an authority signature key

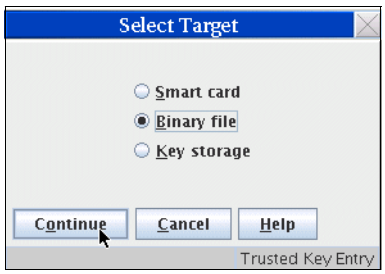


Figure 7-56 Select where the signature key will be stored

Previously, the authority signature key length was always 1024 bits. Now the length can be higher, thus increasing security, but the TKE smart card is limited to 2048 bit keys. Note also that CEX2C accepts only 1024-bit keys. CEX3C accepts all key lengths. See Figure 7-57.

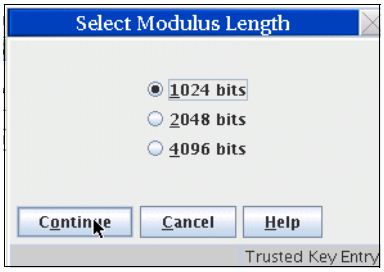


Figure 7-57 Select the signature key modulus length

When the new authority signature key is generated, the Key Saved status window also shows key identifier information (Figure 7-58).

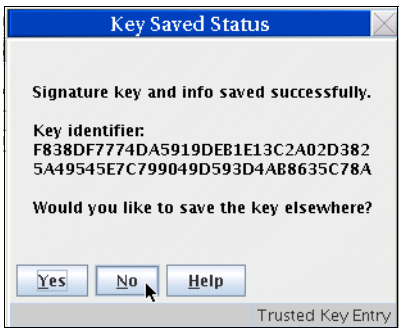


Figure 7-58 Signature key identifier information

In the authority creation process, the generated signature key length and also the key identifier information are also shown in the window (Figure 7-59).

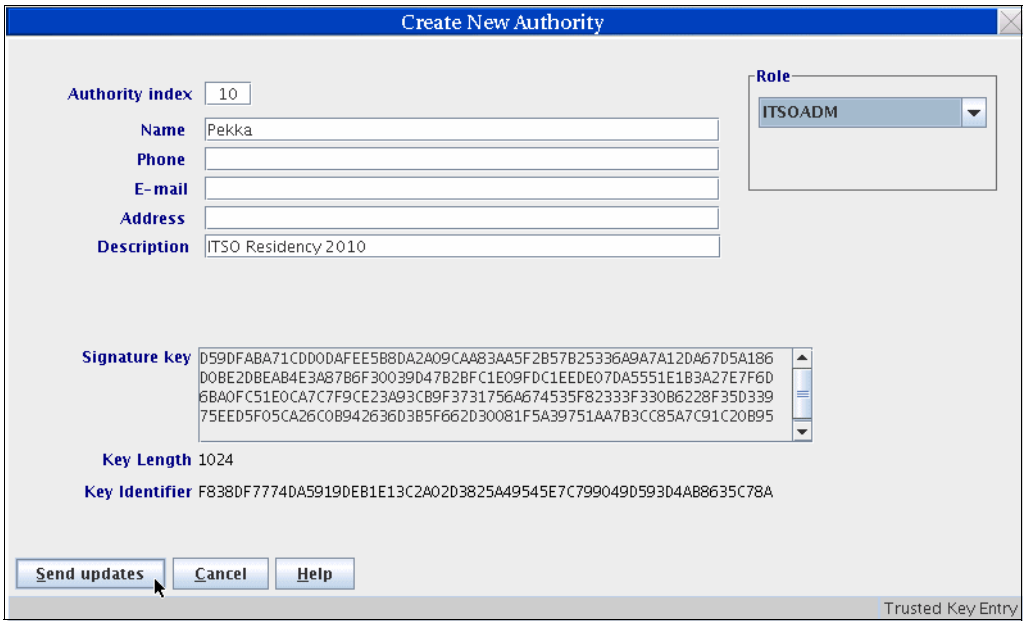


Figure 7-59 Signature key modulus length and identification

Clearing the master key register values

It has been possible to clear the new master key register content already in the earlier TKE versions. This function has been enhanced to be able to clear the old master key register (Figure 7-60). This new capability will also affect the role definition where the corresponding access control point must be marked (Figure 7-60).

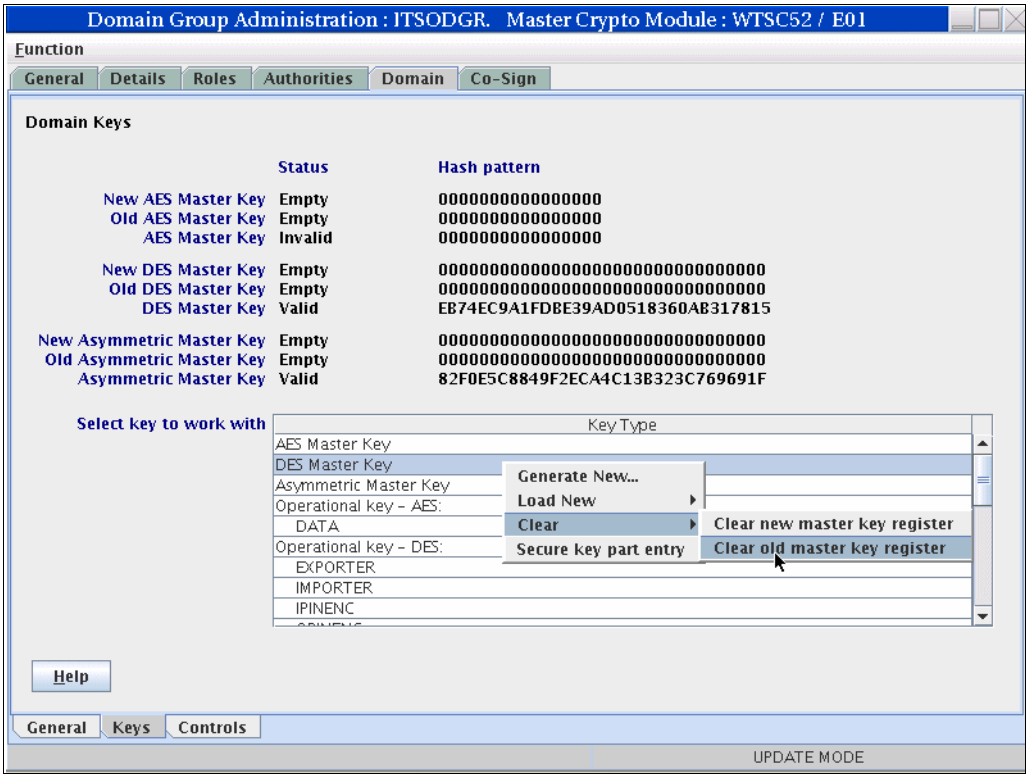


Figure 7-60 Ability to clear the old master key register

TKE main panel

There is a new option to display the loaded signature key information. This is located on the TKE application main panel. This was not possible on the earlier TKE releases.

To view an already loaded signature key, select **Function** and then **Display signature key information** (Figure 7-61). The information displayed about the loaded signature key contains key identification and authority index (Figure 7-62).

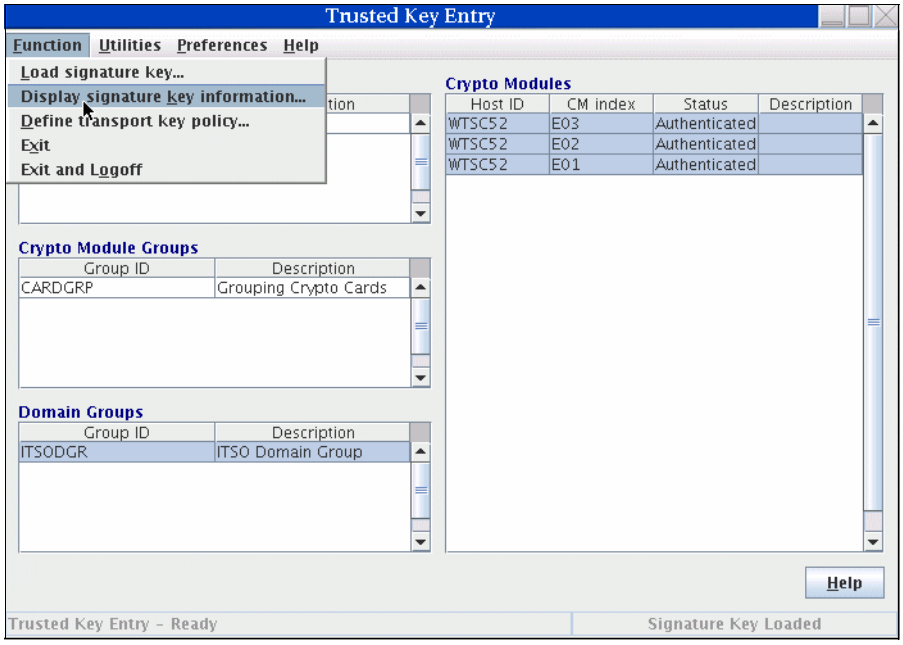


Figure 7-61 Display currently loaded signature key information

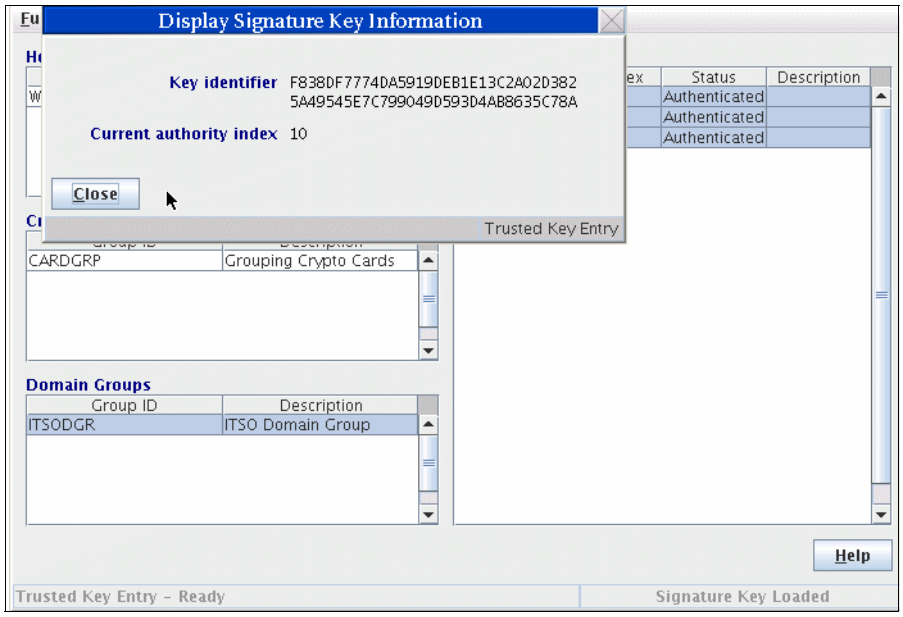


Figure 7-62 Loaded signature key information

7.3.6 Other items

In this section we provide information about other new features that are considered to be useful when using a Trusted Key Entry workstation.

Manage print screens

Since TKE4.2 there have not been any tools or mechanisms to make screen copies for documentation purposes, thus creating problems when writing usage instructions to company personnel.

This missing feature has now been introduced through the Manage Print Screen option, which can be found in the Service Management workspace (Figure 7-63).

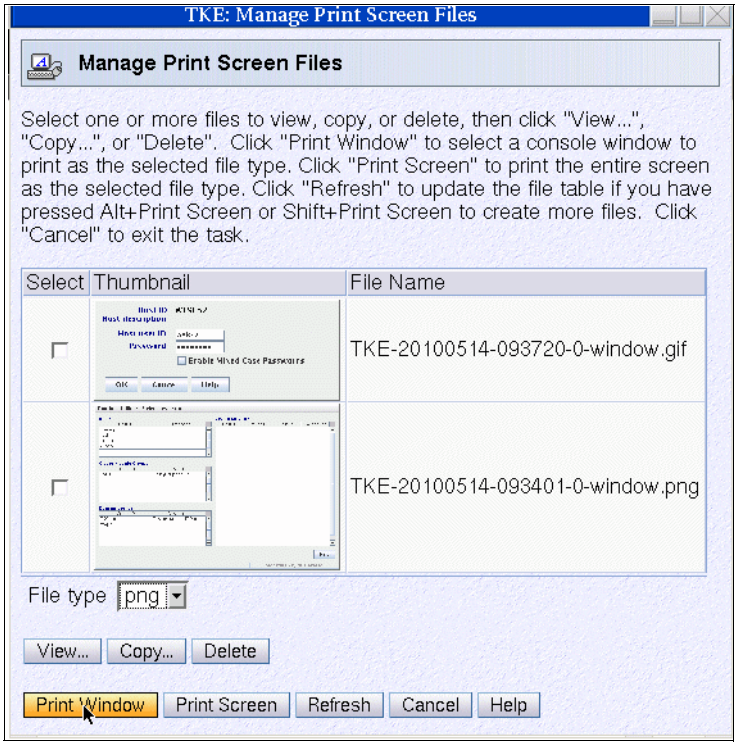


Figure 7-63 Select to create a window copy

This feature is able to save screen shots and also window shots into the TKE hard disk in `.jpg` or `.png` format. You can view the taken screen or window shots by selecting the target shot and then clicking **View**. Unwanted screen shots can be deleted.

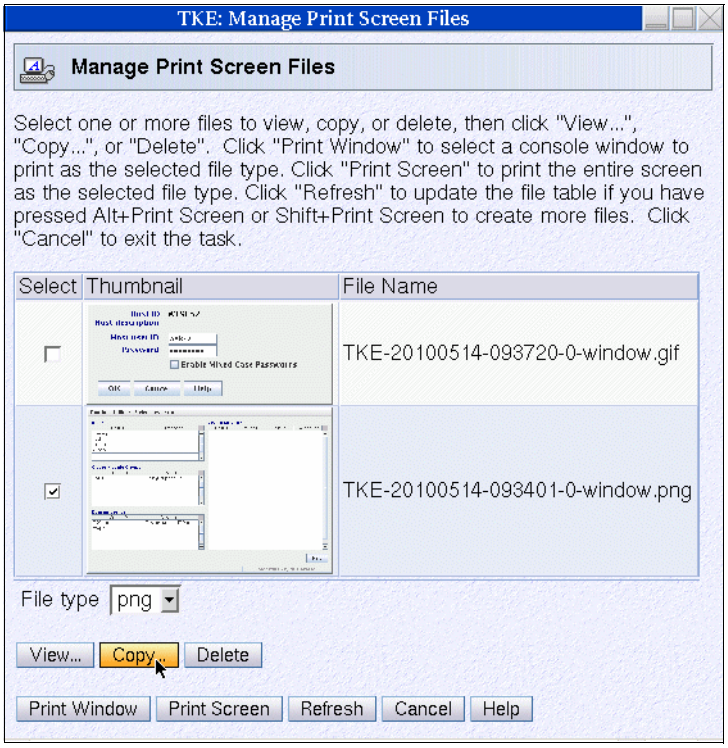


Figure 7-64 Copy the selected print screens to target media

To copy screen shots, scroll down the list of screen shots and select those that you want to copy to an external media device, and then click **Copy** (Figure 7-64).

When clicking Copy, a new window displays on which the external media device can be selected. The options are:

- Diskette
- DVD-RAM
- USB memory

USB memory displays only if one is attached to the TKE workstation. In our case, as in Figure 7-65, we did not have USB memory attached, and so it is not shown in this select list.

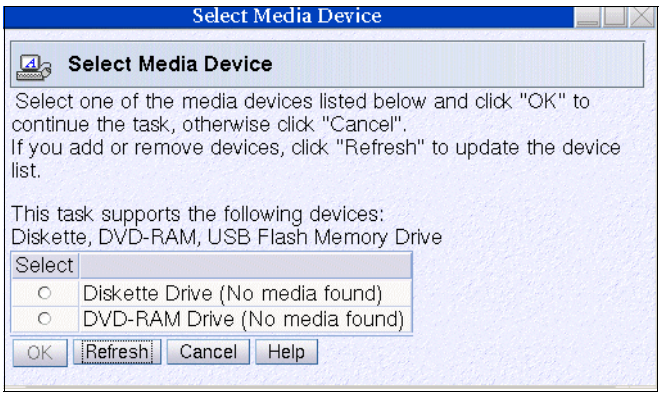


Figure 7-65 Select media for the print screens

NTP

When setting the date and time to the TKE workstation, the NTP protocol is also supported to synchronize time. The process flow is shown in the following figures.

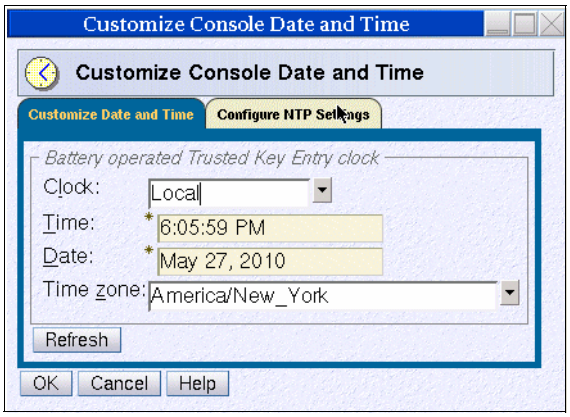


Figure 7-66 Set the time and date for the TKE workstation

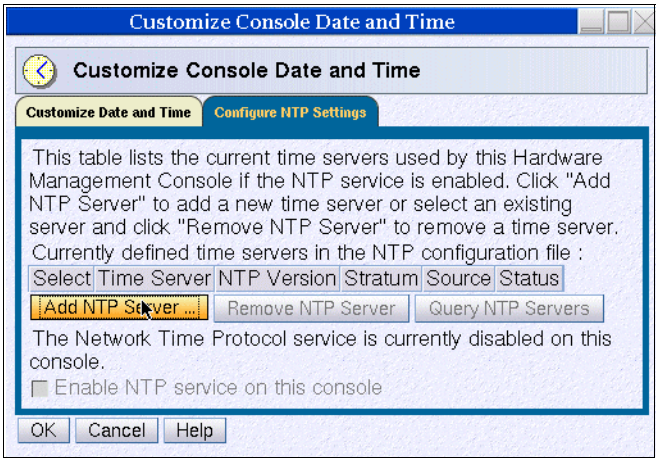


Figure 7-67 Set the TKE workstation time through NTP

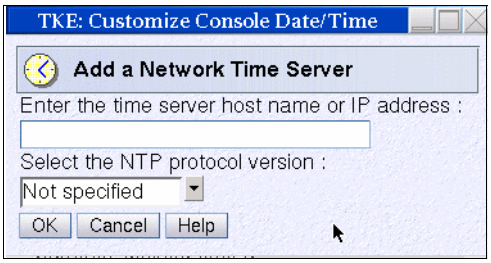


Figure 7-68 Define NTP information

IPv6

TKE workstation now supports IPv6 host addresses. This feature is valid using TKEV5.2 and later with ICSF release HCR7720 and later. You need to install APAR OA23811.

7.4 Migration from previous releases

MIgrating from TKEV5.x is basically a TKE code update. The same physical TKE workstation can be used and also the same smart card readers and smart cards can be used with the TKEV6.0.

We suggest that when migrating from TKEV5.x to TKEV6.0 that you let your IBM CE do the TKE code installation.

If you want to make a backup copy of the important files, use the file management utility and copy the following to the formatted DVD-RAM:

- All profiles that you have created (on the CNM data directory)
- `Host.dat` and `Group.dat` files (on the TKE data directory)

The standard approach is not to change TKE workstation roles, so there is no need to copy them. More details about the migration issues can be found in the TKE manual.

8

A brief overview of integrated hardware cryptography implementation in Linux for System z

This chapter provides an overview of the integrated hardware cryptography infrastructure that allows Linux for System z applications to invoke hardware cryptographic services using clear or secure keys.

A specific focus is given to the secure key cryptographic services support, as provided by the IBM Common Cryptographic Architecture (CCA) API support. In this chapter we refer to the API and libraries as delivered in the CCA Support Program (commonly called CCA host library) Release 4.0.0, which provides:

- Support for the CEX3C coprocessor cryptographic services

  Most of the supported services are the same as the ones supported in z/OS ICSF and include the support already available on the System i® platform, Linux for the System x® platform, and AIX for the 4764 and 4765 Cryptographic Coprocessors.
- Support for CPACF functions, including support for the protected key when running on a System z10 GA3 with a CEX3C coprocessor

For further details about the provided services specifications and administrative functions, see *Linux for System z - Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Gu*ide, SC33-8294.

Earlier implementations of hardware cryptography support in Linux for System z have already been discussed in the following IBM Redbooks publications:

- *Security for Linux on System z*, SG24-7728
- *zSeries Crypto Guide Update*, SG24-6870

- *Using Cryptographic Adapters for Web Servers with Linux on IBM System z9 and zSeries*, REDP-4131
- *Monitoring System z Cryptographic Services*, REDP-4358

8.1 A reminder about Linux for System z cryptographic features

Linux for IBM System z has implemented the necessary support to exploit the System z integrated hardware cryptography features with an objective of giving Linux users the opportunity to host applications with security requirements ranging from securing web traffic up to providing adequate security for sensitive financial transactions.

Early implementations of the support dealt exclusively with RSA clear key operations as provided by the S/390®, zSeries, and System z PCI coprocessors or accelerators and aimed at decreasing the cost of the SSL handshakes as incurred by http communications. The support grew in sophistication so that Linux applications can now use several APIs, including the IBM CCA API, to achieve data confidentiality, data integrity, and authentication using clear keys or secret keys.

As for other operating environments that implement hardware cryptography support, the expected benefits are:

- The saving of applications' MIPS by off-loading the cryptographic workload to specialized hardware coprocessors
- An improved performance induced by the use of specialized engines
- An increased security for highly sensitive items such as secret keys, as they can be physically protected within the secure coprocessors

8.1.1 The cryptographic options available with Linux for System z

Support is provided for the use of the following cryptographic hardware facilities:

- The Crypto Express3 feature in coprocessor or accelerator mode: This assumes that a coprocessor domain has been assigned to the Linux logical partition or to the host z/VM logical partition that executes the Linux guest virtual machine. Historically, support for accelerator mode (that is, clear RSA keys only operations) has been implemented via several APIs that we briefly describe below. Support for the CCA API came later and requires you to have access to the Crypto Express3 coprocessor configured in coprocessor mode.
- The CP Assist for Cryptographic Functions: The CPACF support has been enhanced to support the protected key feature of the CPACF when executing in a System z10 GA3 or follow-on.

Over time, the software support infrastructure develops up to providing the APIs and support libraries configuration shown in Figure 8-1.

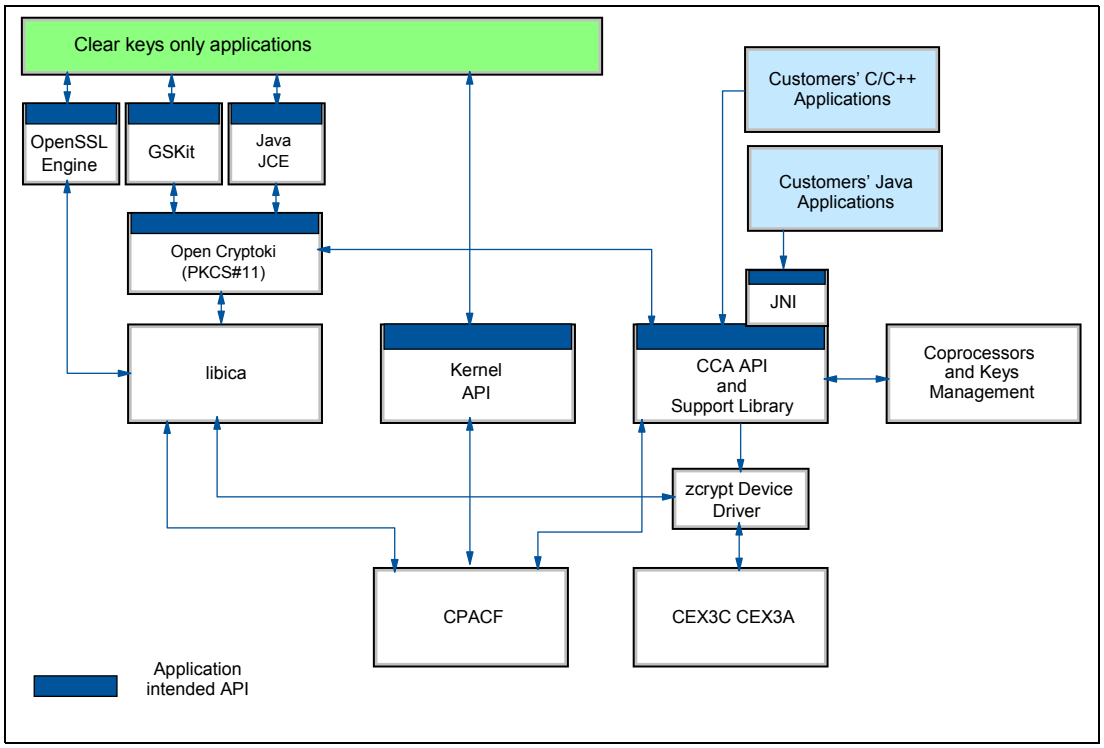


Figure 8-1 Linux for System z cryptographic APIs and support libraries configuration

The cryptographic services that applications request can fall roughly in to either one of the two following categories:

- The clear-keys-only operations, where applications request acceleration only of cryptographic algorithms, without the extra security that secure coprocessors can provide. These are shown on the left of Figure 8-1. Hardware assist is given from the Crypto Express3 coprocessors, either in coprocessor or accelerator mode, for RSA acceleration and from the CPACF for symmetric encryption or decryption or SHA computation.
- The secure key operations as provided by the CCA callable services (or *verbs*) exploiting the Crypto Express3 coprocessor operating in coprocessor mode and the CPACF for clear keys, or protected keys, symmetric encryption and decryption, and SHA computations.

Note the Linux for System z zcrypt device driver (/dev/zcrypt) in charge of interfacing with the Crypto Express3 coprocessor.

8.1.2 Clear keys acceleration support libraries and exploiters

Figure 8-2 focuses on the support infrastructure for the acceleration of clear keys operations as required by early implementations of web applications in Linux for System z.

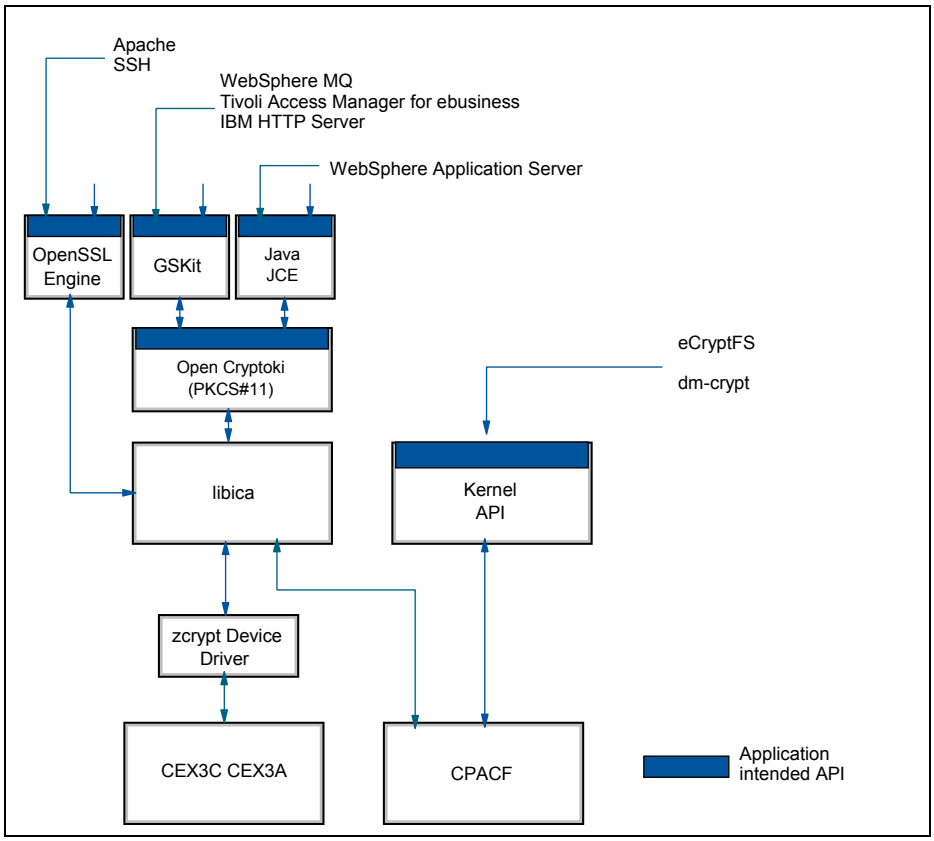


Figure 8-2 Support libraries for clear key operations

Note: In the context of clear-key-only operations, applications are in charge of managing the keys that they use and whenever appropriate use storage facilities such as password-protected key database files.

APIs that use the libica library

The libica library was developed by IBM (ica stands for IBM Cryptographic Adapter) and integrated into the Linux open source. It is intended to provide a low-level API for clear key cryptographic services that can be performed by:

- Software only, as implemented into the libica code and called whenever a similar hardware service is not available
- Calls to the Crypto Express3 coprocessor clear keys functions through the zcrypt device driver
- Direct calls to the CPACF

The following APIs use the libica library:

- The OpenSSL API: This API was initially implemented to be exploited by the Apache web server on Linux to provide RSA hardware assist for the SSL handshake phase of https communications. The OpenSSL API is also called by Secure Shell (SSH) for Linux.
- The PKCS#11 API, as implemented in its Cryptoki open source version: This API provides an intermediate layer for two higher level APIs:
  - The IBM Global Security Key interface tool (GSKit): The IBM proprietary API was ported from other IBM Linux platforms where it was already exploited by products such as WebSphere MQ, Tivoli® Access Manager for ebusiness, and the IBM HTTP server
  - The Java Cryptographic Extensions (JCE) API, with the IBMPKCS11Impl provider

The Linux Kernel's API

This is an API that is part of the Kernel internal components that require cryptographic services. It is built into the kernel as loadable modules, based on kernel compilation options and installed with the kernel. It provides software services, and beginning with Kernel 2.6 it can call the CPACF whenever appropriate for execution of DES, T-DES, AES symmetric encryption and decryption, SHA computation, and pseudo random number generation.

As of the writing of this book, the following applications are known to exploit the Kernel's API and thus have the benefit of hardware cryptography:

- eCryptFS is a set of functions that are an add-on to a kernel-resident file system. It allows the symmetric encryption of files on a per-file or a per-directory basis.
- dm-crypt is a so-called *device-mapper target* that provides symmetric encryption of block devices transparently to user applications.

8.2 The IBM CCA support library and its exploiters

The CCA API provides language bindings for C/C++. It is a synchronous API in that the program execution is held until completion of the requested service is thread-safe and supports concurrent calls from different execution threads

The CCA library and API are implemented via the CCA Support Program for Linux for System z, with a default distribution location of /usr/lib64/libcsulcca.so.

As of the writing of this book, the CCA Support Program is at Release 4.0.0.

8.2.1 The support infrastructure

Figure 8-3 focuses on the cryptographic libraries infrastructure that provides CCA-based, cryptographic services.

r

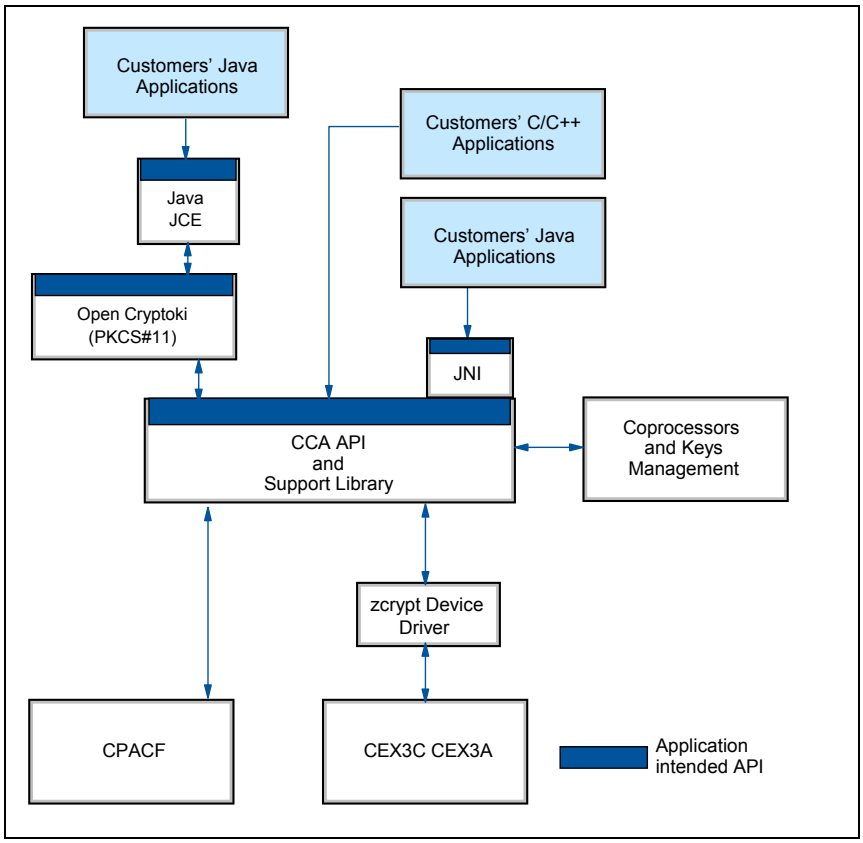


Figure 8-3 CCA support library and API infrastructure

A limited amount of PKCS#11 services, and therefore the equivalent JCE services (using the IBMPKCS11Impl provider), provide secure key services by getting secure coprocessor assist through the CCA support program for DES, T-DES, and RSA:

- Key generation
- Data encrypt and decrypt

A specific Java Native Interface (JNI) is implemented with custom written C code to provide Java applications with direct access to CCA services. This API allows you to simplify the implementation of CCA calls in Java applications without having to go through the JCE stacked service providers and preference list model. Calls for the JNI are also documented in the *Linux for System z - Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide,* SC33-829.

These versions of Java are supported for the JNI:

- Java 1.6.0 for Red Hat Enterprise Linux (RHEL)
- Java 1.4.2 for SUSE Linux Enterprise Server 10 and 11 (SLES 10 and 11) from Novell

A specific support is given to the management of secure keys and secure coprocessors that we discuss in 8.4, "Management of the CCA application's secure keys" on page 238, and 8.5, "Management of the Crypto Express3 coprocessor" on page 240.

Note: Contrary to the z/OS implementation where ICSF transparently allocates the Crypto Express3 to applications request execution threads, the Linux CCA implementation allows users to explicitly allocate or deallocate a specific coprocessor to a process or an execution thread. Refer to the Cryptographic Resource Allocate (CSUACRA) and Cryptographic Resource Deallocate (CSUACRD) services in the reference documentation.

8.2.2 The services provided by the Crypto Express3 coprocessor

As for ICSF in z/OS, the CCA API in Linux for System z gives access to the services provided by the Crypto Express3 coprocessor, as listed below. *Secure keys* refers, as for the z/OS implementation, to keys encrypted under one of the coprocessor master keys (that is, the DES-MK, the ASYM-MK, or the AES-MK).

Again, details on all the CCA services available in the CCA Support Program are described in *Linux for System z - Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide*, SC33-8294.

Data protection using symmetric algorithms

The functions below are performed using secure keys only in the CEX3C coprocessor:

- Data encryption and decryption
  - Using the DES or Triple-DES algorithm
  - Using the AES algorithm
- DES and AES key generation, importation, and distribution

  The key generation can be random generation or based on a derivation algorithm. The symmetric key can be exported in a format compatible with remote devices such as ATMs for an automated remote loading of the key.

Data integrity using symmetric and one-way algorithms

Data integrity support includes:

- Message Authentication Code (MAC), Modification Detection Code (MDC), and one-way hash algorithms are provided. The MAC services require the use of secure keys.
- SHA-1 and SHA-2 (SHA-256, SHA-384, and SHA-512) support.

Asymmetric algorithm support

As of the writing of this book, the CEX3C coprocessor provides only RSA hardware support to System z users through the CCA API. There is, however, an internal implementation of the Diffie Hellman algorithm that is reserved for the coprocessor's own use when securely communicating with the TKE workstation.

Note also that the CCA API refers to Public Key Algorithm (PKA) services to generally designate the RSA services that the CEX3C coprocessor provides.

Depending on the function invoked, the RSA secret private key is required to be presented in secure key form or in clear text form.

As of the writing of this book, the maximum RSA key length supported by the CEX3C feature is 4096 bits. Symmetric key support includes:

- Support of the RSA key pairs management
  - Importation of public-private key pairs in clear text and externally encrypted forms
  - Key pair generation
  - RSA key format translation from 4765 format to smart card format
- RSA digital signature function

  These services require you to use a secure RSA private key:
  - Generation, with selection of the signature formatting method
  - Verification
- Symmetric key encryption function

  The CEX3C coprocessor can proceed with the encryption of secure symmetric keys (DES, Triple-DES, or AES) by an RSA public key, for secure export of these keys, or by an RSA private key when importing such a secure key.

  A specific function is also implemented to assist the symmetric key wrapping or unwrapping during the SSL/TLS protocol handshake with formatting of the resulting symmetric key block.
- Modular exponentiation

  This function is a specific form of the symmetric key encryption function that programs can invoke whenever such an exponentiation has to be applied against a data string. This can be used, for instance, for assisting software implementation of asymmetric algorithms other than RSA that also rely on modular exponentiation.
- Support of RSA retained keys

  Specifying RETAINED is an option when invoking and generating RSA key pair generation by the coprocessor. The key pair is generated in the secure physical boundary of the coprocessor, and the private key is never to leave this physical boundary. The retained private key, when needed, is designated by a label and retrieved by the coprocessor from its protected internal memory.

  Note: Although RSA retained keys are supported in the CEX3C coprocessor, their use in System z is not recommended by IBM. The retained key technologies are intended to be used on other platforms hosting their own version of the IBM 4765 coprocessor.

Financial services support

These services require the use of secure keys:

- Personal identification number (PIN) processing services: Typically generation, verification, and translation, with support for different formats of the PIN block
- Credit card verification value or card security code processing: Typically generation and verification, with support for different card types and algorithms
- Support for secure messaging as implemented in the Europay - Master Card-Visa (EMV) protocol
- Support for composing/decomposing data blocks in the Secure Electronic Transfer (SET) protocol

Other functions supported

The Crypto Express3 hardware also implements a high-quality and high-speed random number generator.

Implementation of customized functions in the coprocessor

Users can customize the functions provided by the CEX3C coprocessor by creating a User Defined Extension (UDX) firmware module that performs the user-designed function as a part of the coprocessor firmware. The creation of UDX modules requires the use of a specific toolkit owned by IBM or approved business partners.

8.2.3 Access control for the CCA services

The CEX3C coprocessor exploits the Access Control Points (ACP) mechanism for the purpose of controlling access to functions that it provides. ACPs are logical switches in the coprocessor firmware that enable or disable the use of specific functions for a given user role. CEX3C coprocessor operations that are executed on behalf of Linux users are executed under the DEFAULT role and are submitted to the access control points being set for this role.

The access control points can be acted upon via the TKE workstation only. See *z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User's Guide* for information about access control points and how they can be managed.

8.3 Supported configurations

The System z hardware cryptographic coprocessors are accessible to Linux applications under different possible logical partition configurations and combinations in the host system, on the basis of cryptographic domains allocated in the logical partitions' image profiles:

- A mixed configuration of z/OS and Linux partitions sharing the cryptographic coprocessors on the domain basis.
- A mixed configuration of Linux instances running as guests in a z/VM logical partition along with other separate z/OS logical partitions. z/VM Control Program funnels the accesses to the coprocessor domains allocated to the logical partition to the Linux guests
- A Linux-only system, with the Linux instances running either native in the logical partitions or as z/VM guests.

We will see in 8.5, "Management of the Crypto Express3 coprocessor" on page 240, that sharing the host system with a z/OS logical partition that executes ICSF can ease the coprocessor administration tasks for Linux users.

8.4 Management of the CCA application's secure keys

An application-independent key storage system is provided with the CCA Support Program and can be used to manage storage records for AES key records, DES key records, and PKA key records:

- AES key storage records hold null and internal AES key tokens.
- DES key storage records hold null, external, and internal DES key tokens.
- PKA key storage records hold null PKA key tokens, and both internal and external public and private PKA key tokens.

We suggest that this key storage system be used, as opposed to having applications managing key tokens in their own local files, to simplify management functions such as key storage backup and key storage re-encipher consequently to a master key change.

It is up to the installation or application development teams to decide on the name of the key storage data and index files, the requirement being that the files be in the `/opt/IBM/CEX3C/keys` directory and the key storage file names are kept in environment variables. There can be different key storage files for different application environments.

8.4.1 Key storage initialization

The key storage must be initialized before any records are created. A key storage initialization CCA service (CSNBKSI) is available to initialize a key storage file using the current symmetric or asymmetric master key. The initialized key storage does not contain any preexisting key records.

Key storage initialization can also be driven using the PCLI utility that is described in 8.5.2, “Using the Command Line Interface panel” on page 241.

8.4.2 Key tokens management

Before a key token can be stored in key storage, a key storage record must be created using the AES Key Record Create, DES Key Record Create, or PKA Key Record Create verb.

The CCA support provides API functions to allow application programs to reference key tokens stored in the key storage files by key label name.

8.4.3 Key storage re-encipher

The key token change callable services (CSNBKTC and CSNDKTC) are available for applications to proceed with key token re-encipher when a master key has been changed.

The procedure to follow is explained in *Linux for System z - Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide*, SC33-8294.

Key storage re-encipher can also be driven using the PCLI utility described in 8.5.2, “Using the Command Line Interface panel” on page 241.

8.4.4 Host-side key caching

The CCA support program provides applications with the capability of caching key records obtained from key storage within the CCA host code. However, the host cache is unique for each host process. If different host processes access the same key record, an update to a key record caused in one process does not affect the contents of the key cache held for other processes.

Caching of key records within the key-storage system is under the control of the CSUCACHE environment variable value and can be suppressed so that all processes access the most current key-records.

8.5 Management of the Crypto Express3 coprocessor

There are three approaches for administering coprocessor domains used by Linux applications. Installations have to choose the solution that best fits their enterprise environment and provides them the flexibility that they need to meet their security policy. As shown in Figure 8-4, the Crypto Express3 coprocessor can be managed from:

- A TKE workstation that has TCP/IP connectivity to the TKEC (TKE catcher) daemon.
- A local Linux utility PCLI.
- The coprocessor management ISPF panels of an ICSF instance running in a z/OS logical partition (provided that the coprocessor domain is temporarily assigned to the z/OS logical partition for the time required to perform the management operations)

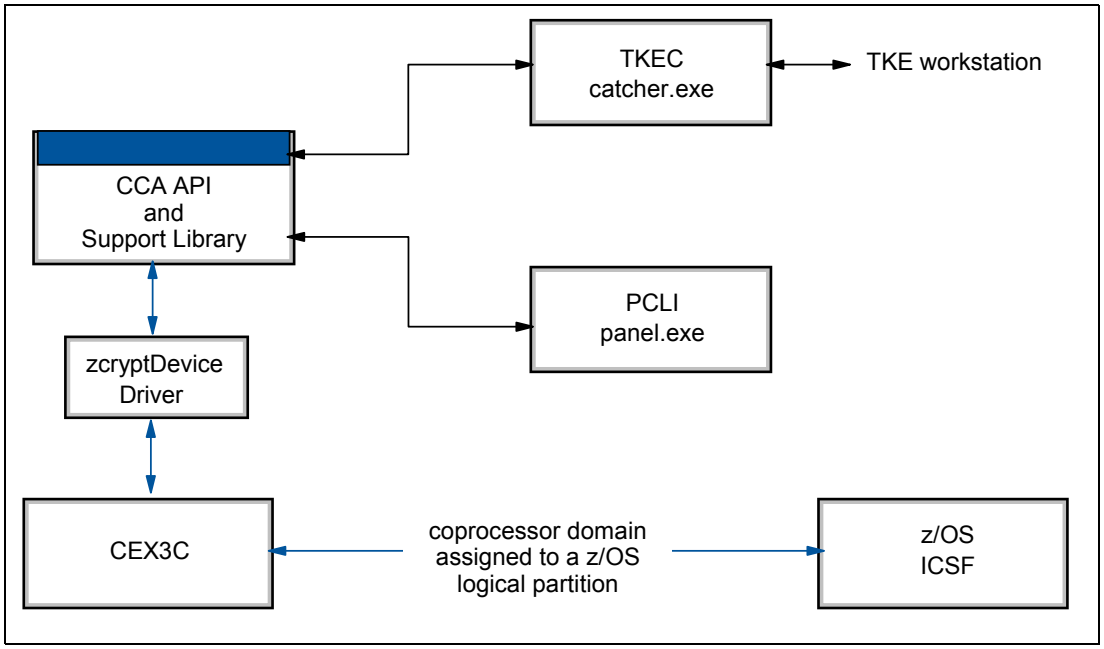


Figure 8-4 Crypto Express3 coprocessor management

8.5.1 Using the TKE workstation

The Trusted Key Entry (TKE) catcher daemon is used to interface with the TKE workstation. This daemon listens on a single TCP/IP port for management communication (by default, port 50003).

The user must have a TKE V6.0 workstation to supported the Crypto Express3 coprocessors. For more information about the TKE workstation, see *z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User's Guide*, SA23-2211.

Note: Managing the Crypto Express3 coprocessor via a TKE workstation is required to modify the access control points in the coprocessor DEFAULT role (that is, to modify the access controls to coprocessor services by Linux users).

8.5.2 Using the Command Line Interface panel

The `panel.exe` utility provides a Linux native mechanism for administering and initializing certain characteristics of active cryptographic coprocessors. It is intended as a basic administration tool for Linux-only System z configurations, where a TKE solution is not available.

The `panel.exe` utility can be used to:

- Determine whether a TKE is currently able to administer a specific active coprocessor.
- List the labels and key types for all the keys in a designated key storage file.
- List the labels for all of the retained keys (RSA private keys stored in the adapter) in the current domain of the CEX3C.
- List the coprocessors currently active in the Linux system and their master key status.
- Load master key parts to the coprocessor.
- Set a master key that was loaded to the coprocessor.

  Note: `panel.exe` is not designed to change the master keys for all the cards in a group. That is a more sophisticated operation.

- Clear master key parts that were previously loaded to the coprocessor but not yet set or confirmed (this is used for when a mistake in entering master key parts has been detected).
- List serial numbers and master key register states of all active cards running CCA that are visible to this Linux host. The total number of active cards and any errors will also be reported.
- Query the master key verification pattern for any master key register in the current domain.
- Initialize a local host key storage file.
- Re-encipher a local host key storage file. (Use this when the master key has been changed to ensure currency with key storage.)
- List available CPACF functions and whether they are supported in the current system image.

The `panel.exe` utility does not support all administration options, such as access control point manipulation. This and other sophisticated administration options require the use of a TKE workstation.

For security reasons, only a root user (real user ID equal to '0') is allowed to use `panel.exe` to load master key parts or to clear previously loaded master key parts. This is enforced at the shared library level in the implementation of the master key process verb, not in the utility itself. Additionally, only the user who created a set of key storage files or the root user will be able to take actions with respect to those key storage files, based on Linux file system permissions.

The utility is installed by the Linux for System z Cryptographic Coprocessor install package or RPM to this path in the Linux system:

```
/opt/IBM/CEX3C/bin/panel.exe
```

Note: For mixed z/OS and Linux configurations, we suggest that administration be accomplished using the z/OS TSO panels as described in the *z/OS ICSF Administrator's Guide*, SA22-7521-14.

8.6 Specifics of CPACF support

The implicit use of the CPACF by cryptographic applications that call the CCA API can be controlled via environment variables and access control points, as explained below. The users can thus decide whether they want to go with the high performance provided by the CPACF or strictly use only the secure services provided by the Crypto Express3.

8.6.1 Environment variables that affect CPACF usage

In the context of CCA services, the CPACF can be used for:

- Providing hardware acceleration for clear key encryption or decryption and for the MDC and SHA computations. This is under control of the CSU_HCPUACLR environment variable.
- Supporting and exploiting the CPACF protected key capability. This is under the control of the CSU_HCPUAPRT environment variable.

The CSU_HCPUACLR environment variable

By default, the CPACF is used for clear key AES encryption and decryption and for MDC generation. In addition, CPACF is the default for the following hash algorithms:

- SHA-1
- SHA-224
- SHA-256
- SHA-384
- SHA-512

Setting this variable to any other value than 1 results in disabling the use of the CPACF for these operations, which are then performed exclusively within the Crypto Express3 card.

The CSU_HCPUAPRT environment variable

This specifies the capability of using the CPACF protected keys. By default, the use of CPACF protected keys is not enabled. Setting the environment variable to 1 enables the use of CPACF protected key for AES and DES, TDES, and MAC.

8.6.2 Access control points that affect CPACF protected key operations

There are two access points that enable the protected key feature:

- Symmetric Key Encipher/Decipher - Encrypted DES keys

  This ACP is ON by default. It enables translating DES keys for use with the CPACF. Without this bit set to ON, the call to the CEX3C to rewrap the key under the CPACF wrapping key fails.

- Symmetric Key Encipher/Decipher - Encrypted AES keys

  This ACP is ON by default. It enables translating DES keys for use with the CPACF. Without this bit set ON, the call to the CEX3C to rewrap the key under the CPACF wrapping key fails.

8.6.3 Overview of the CPACF protected key operations in Linux for System z

The CPACF protected key facility operates the same as in z/OS (as explained in 3.4, “Protected key theory” on page 59), exploiting wrapping keys generated at LPAR activation, with software layers specific to the Linux implementation being involved.

The CPACF exploitation layer examines the cryptographic calls received from applications to see whether they can be redirected to the CPACF. If so, this layer makes preparations (including translating secure keys to protected keys), and then calls the CPACF directly.

The device driver and the other layers are used for protected key support, for translating keys transparently to the application. After translation, the translated-key is stored in an invisible runtime cache so that the next use of the key can avoid the translation step.

For protected key usage, a CEX3C feature must be available and allocated for use by the thread.

Important: The CPACF is an independent hardware unit, like the CEX3C itself, and can be independently configured as available or unavailable while a Linux instance is running by service technicians performing service actions. If the CPACF is cycled, it will generate a new wrapping key for translated keys, invalidating all of the keys in the CCA library key translation cache. Therefore, it is never advisable to attempt such a service action while there are system instances with applications running that use the CPACF.

If such an action is undertaken, applications should be stopped and restarted so that libcsulcca.so is unloaded from memory and reloaded. This causes the key cache to be cycled.

8.7 Performance data

Although this has not been done yet for the System z10 GA3, documents available on this site usually include a chapter on Linux for System z hardware cryptography performance, along with a more complete report on z/OS cryptography performance. The reader is advised to go to this site and check what has been made publicly available.

8.8 Download sites and Linux distributions CCA support

The following websites offer valuable information and downloads:

- For the Crypto Express3 coprocessor characteristics:

  `http://ibm.com/security/cryptocards/pciecc/overview.shtml`
- For the *Crypto Express3 Programmer's Guide*:

  `http://ibm.com/security/cryptocards/pciecc/library.shtml`
- For a no-cost download of the CCA support library:

  `http://ibm.com/security/cryptocards/pciecc/ordersoftware.shtml`

Linux for System z distributions with CCA secure key support

The following Linux distributions for System z provide CCA secure key support:

- Novell - SUSE Linux Enterprise Server SLES 10 SP1
- RedHat Enterprise Linux
  - RHEL 5.1 - Device Driver
  - RHEL 5.2 - PKCS#11 support

9

Using Elliptic Curve Cryptography (ECC) on z/OS

This chapter describes a short program that calls the ICSF PKCS#11 services to generate a pair of ECC keys, then signs a string of data using the private key and verifies the signature using the public key. We first provide practical information about ECC.

9.1 Elliptic Curve Cryptography in a nutshell

As the name implies, this section is intended to be a very high-level introduction to Elliptic Curve Cryptography. ECC is a relatively new field in cryptography with practical implementations that surfaced in the late 1980s, which is approximately10 years after the public key cryptography breakthrough materialized by the RSA algorithm.

ECC compares advantageously with its predecessor algorithms such as RSA or Diffie Hellman and is slated to replace them within the next few years.

9.1.1 Another asymmetric algorithm

Cryptographic algorithms and keys based on the mathematical properties of the so-called *elliptic curves* fall in the category of public key cryptography. That is, the cryptographic services users own a key pair, and one key has a publicly known value (the public key), while the other key of the pair is to be kept secret by its owner (the private key). As for other asymmetric algorithms, elliptic curve-based cryptography can be the foundation for cryptographic services such as:

- Digital signature generation and verification
- Data encryption and decryption
- Key agreement

Each of these processes involves the transformation of known data using a permanently established secret value: the *key*. The two first services are widely provided today by the ubiquitous RSA algorithm. The third one is usually provided using the Diffie-Hellman algorithm.

9.1.2 The advantages of Elliptic Curve Cryptography

There are two main advantages that ECC has when compared to other asymmetric algorithms:

- Its improved security: That is how difficult it is to find the value of the secret private key knowing the public key. Typically, asymmetric algorithms use very long keys to ensure that the discovery of the private key value is not practically feasible. Here the mathematical properties of points located on elliptic curves are exploited to design key generation and data encryption algorithms. Whereas the discovery of an RSA private key is based on the capability of factoring large numbers, attacking an ECC algorithm requires finding discrete logarithms of large numbers, themselves generated using the elliptic curve's mathematical properties. The elliptic curve discrete logarithm problem (ECDLP) is believed to be (and to stay for the middle term future) far more difficult to solve than factoring large numbers. A consequence of this is that ECC keys can be made shorter than RSA keys still offering an equivalent "strength" against discovery of the private key value.
- Performance: Although the involved computations remain at approximately the same level of complexity as the RSA ones, dealing with shorter keys speeds up the process of encryption and decryption as measured in CPU, or coprocessor, time.

9.1.3 Mathematical background

We do not dive into the mathematical theory of elliptic curves here, but we stay at a very high and practical level in describing how the elliptic curves properties are exploited to provide proper cryptographic algorithms and related key pairs.

The elliptic curve

An elliptic curve, for the practical use of ECC, is made of points of coordinates that solve the equation `y**2 = x**3 + a x + b`, where a and b are coefficients that affect the shape of the curve and that, among other data, must be shared between parties exchanging information encrypted with ECC. Figure 9-1 shows an example of an elliptic curve.

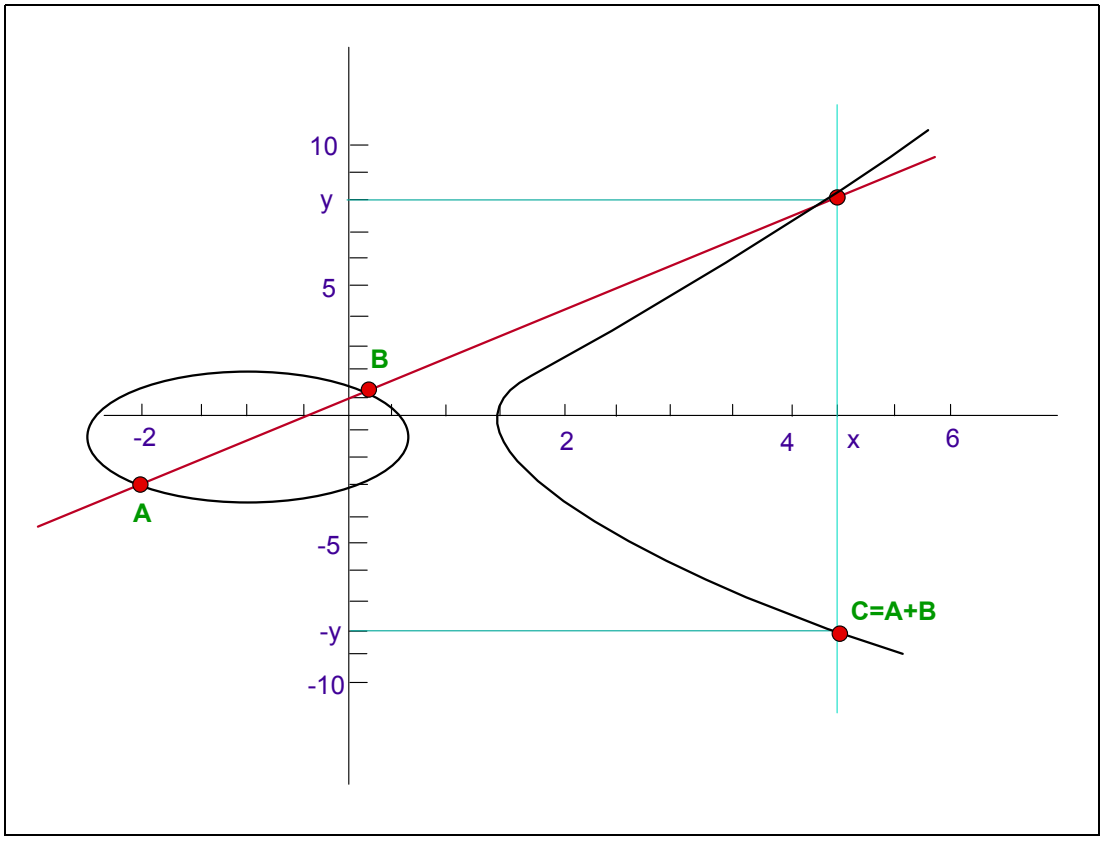


Figure 9-1 Example of elliptic curve

A peculiarity of elliptic curves is that algebraic operations can be defined for points on the curve. That is, the points' coordinates come to play in these operations, but indirectly as these operation's operands are really the points themselves. As an example, in Figure 9-1 the point C is the "addition" of points A and B. As a consequence of being able to define points' "addition," it is also mathematically possible to define the "multiplication" of a point by a scalar (a multiplication is still a series of additions), thus creating the conditions required to contend with the elliptic curve discrete logarithm problem for anybody who wants to reverse the process.

Readers interested in learning more about an elliptic curve's algebraic properties are invited to browse the many articles that cover this topic today on the internet.

9.1.4 ECC in practice

Elliptic curve mathematical properties are exploited to produce ECC key pairs and to proceed with data transformation using these keys. Note that the ECC encryption and decryption processes require that parties exchanging encrypted data share the set of mathematical parameters that define the elliptic curve that is used.

The ECC key pair

The ECC key generation process delivers:

- A random private key.
- A public key, which is the private key “multiplied” by a point chosen on the curve (also known as the *generator* or *base* point G). That is, the public key is a point on the curve. The coordinates of this point will be used automatically for the relevant cryptographic computations.

The ECC algorithms

Different algorithms are used, depending on whether the process is digital signature, data encryption, or key agreement. These algorithms imply using elliptic curve points’ coordinates after scalar multiplication of these points.

Algorithms specifications are published by the Standards for Efficient Cryptography Group (SECG) at:

http://www.secg.org/

Domain parameters

The set of mathematical parameters that describes the curve being used is called the *domain parameters* and is to be agreed upon by both the encrypting and decrypting side. These parameters consist of:

- A curve coefficient a
- A curve coefficient b
- A prime integer p
- The base point G
- The order of base point G
- The so-called cofactor h

Specialized articles on the internet provide details about these domain parameters. For the sake of practical use of ECC, predefined curves (that is, predefined sets of domain parameters) are standardized and can be referred to by users as the curve to be used. This is explained in the next section.

Predefined curves

ECC requires you to specify an elliptic curve's domain parameters. As the "security" of the curve might vary depending on its shape, several curves have been predefined using accepted methods and standards that are recommended for use by various security standardization bodies. There are two prevailing bodies today that endorse sets of elliptic curve recommended for use with ECC:

- USA National Institute of Standards and Technology (NIST): NIST recommends a predefined set of curves for federal government use. Refer to the following website for further details:

  http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf

- The Brainpool Standard Curves and Curve Generation, which are provided by the Internet Engineering Task Force (IETF) in response to perceived shortcomings of the NIST recommended curves. Refer to the following website for further details:

  http://www.ecc-brainpool.org/download/Domain-parameters.pdf

Each of the curves defined by these standardization bodies comes as a set of domain parameters, with a name and an object identifier (OID) allocated by the Internet Assigned Numbers Authority (IANA). As an example, NIST recommends the use of a curve named "secp192r1", which can also be identified using the OID 1 2 840 10045 3 1 1. Note that 192 indicates the length of the key.

Note: Information about OIDs, such as the object that they designate or their ASN.1format, can be found at:

http://www.oid-info.com/

Users of ICSF PKCS#11 services can elect to use ECC based on predefined curves rather than providing their own set of domain parameters. ICSF supports the NIST and Brainpool named curves listed in Table 9-1, still with the numeric value in the name indicating the key length in bits.

Table 9-1 Named curves supported in ICSF

NIST-recommended named curves	Brainpool-defined named curves
secp192r1 – { 1 2 840 10045 3 1 1 } secp224r1 – { 1 3 132 0 33 } secp256r1 – { 1 2 840 10045 3 1 7 } secp384r1 – { 1 3 132 0 34 } secp521r1 – { 1 3 132 0 35 }	brainpoolP160r1 – { 1 3 36 3 3 2 8 1 1 1 } brainpoolP192r1 – { 1 3 36 3 3 2 8 1 1 3 } brainpoolP224r1 – { 1 3 36 3 3 2 8 1 1 5 } brainpoolP256r1 – { 1 3 36 3 3 2 8 1 1 7 } brainpoolP320r1 – { 1 3 36 3 3 2 8 1 1 9 } brainpoolP384r1 – { 1 3 36 3 3 2 8 1 1 11 } brainpoolP512r1 – { 1 3 36 3 3 2 8 1 1 13 }

Attention: It is expected in the PKCS#11 API that the named-curve OID will be provided, encoded in Distinguished Encoding Rule (DER). For example, if we select the curve secp256r1 with a OID {1 2 840 10045 3 1 7}, the DER encoded OID is represented by the hexadecimal string “'06082a8648ce3d030107”.

DER encoding facilities can be found on the internet or as utility programs in certain operating systems. The ASN.1 value of the OID to encode can be found at:

http://www.oid-info.com/

Below we provide the already encoded OIDs for the named curves that z/OS PKCS#11 supports:

/* NIST Curves */

```
secp192r1 = '06082a8648ce3d030101'x         /* {1 2 840 10045 3 1 1} */
secp224r1 = '06052b81040021'x                       /* {1 3 132 0 33} */
secp256r1 = '06082a8648ce3d030107'x         /* {1 2 840 10045 3 1 7} */
secp384r1 = '06052b81040022'x                       /* {1 3 132 0 34} */
secp521r1 = '06052b81040023'x                       /* {1 3 132 0 35} */
/* Brainpool Curves */
BrainPoolP160R1 = '06092b2403030208010101'x /* {1 3 36 3 3 2 8 1 1 1} */
BrainPoolP192R1 = '06092b2403030208010103'x /* {1 3 36 3 3 2 8 1 1 3} */
BrainPoolP224R1 = '06092b2403030208010105'x /* {1 3 36 3 3 2 8 1 1 5} */
BrainPoolP256R1 = '06092b2403030208010107'x /* {1 3 36 3 3 2 8 1 1 7} */
BrainPoolP320R1 = '06092b2403030208010109'x /* {1 3 36 3 3 2 8 1 1 9} */
BrainPoolP384R1 = '06092b240303020801010b'x /* {1 3 36 3 3 2 8 1 1 11} */
BrainPoolP512R1 = '06092b240303020801010d'x /* {1 3 36 3 3 2 8 1 1 13} */
```

Suggested key sizes

As it is recognized that ECC can afford using shorter keys than RSA for the same level of security, the NIST recommendations for minimum key sizes given in Table 9-2 are good examples of how the key lengths compare between the two algorithms. Key sizes on the same row provide approximately an equivalent strength against attacks, as per current knowledge as of the writing of the NIST document.

Table 9-2 NIST recommended key sizes

RSA key size (in bits)	ECC key size (in bits)
1024	160
2048	224
3072	256
7680	384
15360	512

Further details on NIST recommendations can be found at:

http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf

9.2 Our sample program

The program that we describe calls ICSF PKCS#11 services to use ECC for:

- Generating a key pair
- Encrypting test data via the digital signature generation service
- Decrypting test data using the digital signature verification service

We keep the program description at the ICSF calls syntax level and focus on specific call parameters of interest. We do not provide a source code example, as these services can be invoked from several different high-level languages that are commonly used.

As usual, readers can refer to the *ICSF Application Programmer's Guide,* SA22-7522-13, and *ICSF Writing a PKCS#11 Application*, SA23-2231-02, for further details.

9.2.1 Token creation

The CSFPTRC (Token Record Create) ICSF service is called. Figure 9-2 shows the syntax of the service call.

```
CALL CSFPTRC(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             handle,
             rule_array_count,
             rule_array,
             attribute_list_length,
             attribute_list)
```

Figure 9-2 The CSFPTRC ICSF callable service syntax

Required authorization

The calling user ID must be permitted as SO with UPDATE authority to the token to be created.

Rule array

There is one rule array keyword to be specified for the CSFPTRC service. The rule_array_count should therefore be specified as "1", and the keyword that we use is "TOKEN". This triggers the initialization of a token in the TKDS. The name of the token has to be prepared in the handle field, as explained below, and other specifications are given in the attribute list.

Handle

On input, this is a 44-byte empty field, except for the 32-byte name of the token to be created (handles are explained in "The token and object handle" on page 148). The token create service completes the information in the remaining 12 bytes.

The token handle is referred to when it comes to delete the token with the CSFPTRD (Token Record Delete).

If using the System z Assembler Language, the token handle can be defined as:

```
TOK_HDL  DC    CL44'PK.TOKEN4'   TOKEN HANDLE
```

Attribute list

The attribute list for a token is specified in Table 9-3.

Table 9-3 Attribute list

Bytes	Description
0 - 31	Manufacturer ID
32 - 47	Model
48 - 63	Serial number
64 - 67	Reserved - hexadecimal zeros

With System z Assembler Language, the attribute list length and the attribute list can be coded as:

```
ATTR_LEN DC    F'68'
ATTR     DC    CL32'MANUFACTURED BY ITSO POUGHKEPSIE'
         DC    CL16'MODEL 0000'
         DC    CL16'S/N 000000'
         DC    16X'00'
```

Looking at the token after its creation, using the ICSF Token Browser, displays the information shown in Figure 9-3.

```
-------------------- ICSF Token Management - Token Details --- Row 1 to 2 of 2
 COMMAND ===>                                                  SCROLL ===> PAGE

 Token name: PK.TOKEN4
 Manufacturer: MANUFACTURED BY ITSO POUGHKEEPSIE
 Model: MODEL 0000
 Serial Number: S/N 000000
 Number of objects: 0
```

Figure 9-3 Looking at the token after its creation

9.2.2 ECC key pair generation

The ICSF service to be called is CSFPGKP (Generate Key Pair). This service was described in 6.5, “Exploiting the PKCS#11 services” on page 139, with the generation of an RSA key pair.

The syntax of the service call is shown again in Figure 9-4.

```
CALL CSFPGKP(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             token_handle,
             rule_array_count,
             rule_array,
             public_key_attribute_list_length,
             public_key_attribute_list,
             public_key_object_handle,
             private_key_attribute_list_length,
             private_key_attribute_list,
             private_key_object_handle)
```

Figure 9-4 The CSFPGKP ICSF callable service syntax

Required authorization

The calling user ID must be permitted as SO with CONTROL authority, or as USER with UPDATE authority to the token being used.

Rule array

There is no rule array keyword to be specified for the CSFPGKP service. The rule_array_count should therefore be specified as “0”.

Public key attributes

The attributes that we want to give to the public key are:

- CKA_CLASS with a value of “CKO_PUBLIC_KEY:
- CKA_KEY_TYPE with a value of “CKK_EC”
- CKA_EC_PARAMS

  This is the identification of a predefined set of domain parameters.

  Attention: It is expected that the named-curve OID will be provided here, encoded in DER. In our example we selected the curve secp256r1, with a OID {1 2 840 10045 3 1 7}. The DER encoded OID is represented by the hexadecimal string “'06082a8648ce3d030107”.

The public key attribute list

Taking as an example a program coded in System z Assembler Language, the attribute list and its length would be:

```
PUB_ATTL DC    AL4(PUBLEND-PUBLSTRT)   ATTRIBUTE LIST LENGTH
PUBLSTRT EQU   *
PUB_ATT  DC    X'0003'           NUMBER OF ATTRIBUTES
*
         DC    X'00000000'       CKA_CLASS
         DC    X'0004'          LENGTH OF VALUE
         DC    X'00000002'      CKO_PUBLIC_KEY
*
```

```
         DC    X'00000100'       CKA_KEY_TYPE
         DC    X'0004'           LENGTH OF VALUE
         DC    X'00000003'       CKK_EC
*
         DC    X'00000180'       CKA_EC_PARAMS
         DC    X'000A'           LENGTH OF VALUE
         DC    X'06082a8648ce3d030107' secp256r1 - MixeD CASE
PUBLEND  EQU   *
```

Private key attributes

The attributes that we want to give to the private key are:

- CKA_CLASS with a value of “CKO_PRIVATE_KEY”
- CKA_KEY_TYPE with a value of “CKK_EC”

Note that the private key attributes are described in the *ICSF Writing PKCS#11 Applications,* SA23-2231-02, and the default attribute values are specified. In particular, the CKA_SIGN attribute has the value “TRUE” by default, meaning that the key can be used for generating digital signature.

The private key attribute list

Taking as an example a program coded in System z Assembler Language, the attribute list and its length are:

```
PRV_ATTL DC    AL4(PRVLEND-PRVLSTRT)   ATTRIBUTE LIST LENGTH
PRVLSTRT EQU   *
PRV_ATT  DC    X'0002'           NUMBER OF ATTRIBUTES
*
         DC    X'00000000'       CKA_CLASS
         DC    X'0004'           LENGTH OF VALUE
         DC    X'00000003'       CKO_PRIVATE_KEY
*
         DC    X'00000100'       CKA_KEY_TYPE
         DC    X'0004'           LENGTH OF VALUE
         DC    X'00000003'       CKK_EC
PRVLEND  EQU   *
```

The object handles

ICSF creates one handle for the private key object and one handle for the public key object.

Looking into the token

Looking into the token after execution of the CSFPGKP service displays the information shown in Figure 9-5.

```
-------------------- ICSF Token Management - Token Details --- Row 1 to 2 of 2
 COMMAND ===>                                                  SCROLL ===> PAGE

 Token name: PK.TOKEN4
 Manufacturer: MANUFACTURED BY ITSO POUGHKEEPSIE
 Model: MODEL 0000
 Serial Number: S/N 000000
 Number of objects: 2

 Select objects to process then press ENTER

 Press END to return to the previous menu.


--------------------------------------------------------------------------------
 _ Object 1         PUBLIC KEY    PRIVATE: FALSE      MODIFIABLE: TRUE
    LABEL:          <Not-specified>
    SUBJECT:        <Not-specified>
    ID:             <Not-specified>
    EC  PARAMS:     Named Curve - secp256r1
    EC POINT:       044104BBB43651D786F3A2B44F165FAD8770BCD12C84C9AB5B7E89C8...
    USAGE FLAGS:    Enc(T),Verify(T),VerifyR(T),Wrap(T),Derive(T)

 _ Object 2         PRIVATE KEY   PRIVATE: TRUE       MODIFIABLE: TRUE
                                  EXTRACTABLE: TRUE   SENSITIVE: FALSE
    LABEL:          <Not-specified>
    SUBJECT:        <Not-specified>
    ID:             <Not-specified>
    EC  PARAMS:     Named Curve - secp256r1
    USAGE FLAGS:    Dec(T),Sign(T),SignR(T),Unwrap(T),Derive(T)
```

Figure 9-5 The test token with the generated key pair

The key objects can be selected to view details. As an example, selecting the private key object yields the display shown in Figure 9-6 on page 256.

```
-------------- ICSF Token Management - Private Key Object Details ------------
 COMMAND ===>                                                 SCROLL ===> PAGE
 Object 2      from token label: PK.TOKEN4

 Select an Action:
   1  Process select DER fields(*) using external command
         Enter UNIX command pathname (see panel help for details):
   2  Modify one or more fields with the new values specified
   3  Delete the entire object
 ------------------------------------------------------------------------------
                                                                        More:     +
    OBJECT CLASS:                  PRIVATE KEY
    PRIVATE:                       TRUE
    MODIFIABLE:                    TRUE
    LABEL:                         <not-specified>
                         New value:
    SUBJECT*:                      <not-specified>

    ID:                            <not-specified>
                         New value:
    KEY TYPE:                      EC
    START DATE:                    <not-specified>
                         New value:               YYYYMMDD
    END DATE:                      <not-specified>
                         New value:               YYYYMMDD
    DERIVE:                        TRUE
    LOCAL:                         TRUE
LOCAL:                           TRUE
KEY GEN MECHANISM:               UNAVAILABLE INFORMATION
DECRYPT:                         TRUE
                     New value: FALSE
SIGN:                            TRUE
                     New value: FALSE
SIGN RECOVER:                    TRUE
                     New value: FALSE
UNWRAP:                          TRUE
                     New value: FALSE
EXTRACTABLE:                     TRUE          (Cannot be changed from FALSE
                     New value: FALSE           to TRUE)
SENSITIVE:                       FALSE         (Cannot be changed from TRUE
                     New value: TRUE            to FALSE)
ALWAYS SENSITIVE:                FALSE
NEVER EXTRACTABLE:               FALSE
 New value: FALSE          to TRUE)
 SENSITIVE:                       FALSE         (Cannot be changed from TRUE
                      New value: TRUE            to FALSE)
 ALWAYS SENSITIVE:               FALSE
 NEVER EXTRACTABLE:              FALSE
 FIPS140:                        FALSE
 APPLICATION:                    <not-specified>
 VALUE:                          Not displayable
 EC PARAMS*: Named Curve - secp256r1
```

Figure 9-6 EC private key object

9.2.3 ECC signature generation

The data are encrypted using a Digital Signature Algorithm (DSA) variant that combines with ECC. Note that for a real signature generation, the data to be encrypted is expected to be a hash value of the data string to be signed. We do not proceed in this simple test program with generation of a hash value. Instead, we perform the digital signature algorithm against the raw data themselves, just for the purpose of proving the correctness of the encryption, then decryption, process.

The ICSF PKCS#11 service call is CSFPPKS (Private Key Sign). Figure 9-7 shows the syntax.

```
CALL CSFPPKS(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             cipher_value_length,
             cipher_value,
             key_handle,
             clear_value_length,
             clear_value )
```

Figure 9-7 The CSFPPKS ICSF callable service syntax

Required authorization

The calling user ID must be permitted as USER with READ authority to the token being used.

Rule array

There is one rule array keyword to be specified for the CSFPPKS service. The rule_array_count should therefore be specified as “1”, and the keyword that we use is “ECDSA” (Mechanism is Elliptic Curve with DSA signature generation).

Cipher value

This is the actual string to be encrypted. As previously mentioned, this is the test data string itself, whereas for a real signature generation it would be expected to be a hash of the data string to be signed.

Clear value

This is the output field that receives the digital signature.

Key handle

This is the handle of the private key object that was created in the previous step.

9.2.4 ECC signature verification

The signature verification consists of decrypting the signature using the public key and verifying that the result matches the data string that was encrypted (that is, for real signature, the hash of the data string to be signed). These operations are performed by the ICSF

PKCS#11 service call CSFPPKV (Public Key Verify), which states whether the signature verifies. Figure 9-8 shows the service call syntax.

```
CALL CSFPPKV(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             clear_value_length,
             clear_value,
             key_handle,
             cipher_value_length,
             cipher_value )
```

Figure 9-8 The CSFPPKV ICSF callable service syntax

Required authorization

The calling user ID must be permitted as SO with READ authority, or USER with READ authority, to the token being used.

Rule array

There is one rule array keyword to be specified for the CSFPPKV service. The rule_array_count should therefore be specified as “1”, and the keyword that we use is “ECDSA” (Mechanism is Elliptic Curve with DSA signature generation).

Cipher value

This is the actual string that was submitted to encryption at signature generation. As previously mentioned, this is the test data string itself, whereas for a real signature generation it would be expected to be a hash of the data string to be signed.

Clear value

This is the digital signature to be verified.

Key handle

This is the handle of the public key object that was created in the key generation step.

9.2.5 Token deletion

Optionally, the token can be deleted by calling the CSFPTRD (Token Record Delete) ICSF PKCS#11 service. Figure 9-9 shows the syntax of the service.

```
CALL CSFPTRD(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             handle,
             rule_array_count,
             rule_array)
```

Figure 9-9 The ICSF CSFPTRD callable service syntax

Required authorization

The calling user ID must be permitted as SO with UPDATE authority to the token to be deleted.

Rule array

There is one rule array keyword to be specified for the CSFPTRD service. The rule_array_count should therefore be specified as “1”, and the keyword that we use is “TOKEN”.

Handle

This is the handle of the token as produced by the CSFPTRC service.

A

Sample programs in assembler and REXX

This appendix provides REXX and assembler code samples that were used in demonstrating the capabilities of CPACF protected keys. Also included is JCL to invoke one of the programs.

There are also REXX programs REXIQF and REXIQA, demonstrating the capabilities of the CSFIQF and CSFIQA interfaces.

Note: The code supplied here has not been subjected to any formal IBM test and is distributed on an *as is* basis without any warranty, either express or implied. The implementation of any of the techniques described or used herein is a customer responsibility and depends on the customer's operational environment. While each item might have been reviewed for accuracy in a specific situation and might run in a specific environment, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

REXPK001 REXX program

This program illustrates how protected key processing can be used and confirms that data encrypted using an AES 256-bit protected key can be decrypted using the same key as a secure key.

```
/* Rexx */                                                                00010000
/*--------------------------------------------------------------------*/  00020000
/*                                                                    */  00030000
/*  Invoke CSNBSYE to encrypt using protected key (from SECURE key) */  00040000
/*  Invoke CSNBSAD to decrypt using SECURE key.                       */  00050000
/*                                                                    */  00060000
/*--------------------------------------------------------------------*/  00070000
                                                                          00080000
/*--------------------------------------------------------------------*/  00090000
/*                                                                    */  00100000
/*   Invoke CSNBSYE                                                   */  00110000
/*                                                                    */  00120000
/*--------------------------------------------------------------------*/  00130000
                                                                          00140000
/* ************************************************** */                  00150000
/* initialize parameters for common use               */                  00160000
/*                                                    */                  00170000
                                                                          00180000
data = 'Bonjour matelot. Comment ca va. Tres bien merci.'                 00190000
key  = 'LENNIES.SECURE.AESKEY01'                                          00200000
                                                                          00210000
/* trace i                                            */                  00220000
say ' ';                                                                  00230000
say ' ';                                                                  00240000
say ' ';                                                                  00250000
say ' ';                                                                  00260000
say '-----------------------------------------';                         00270000
say ' ';                                                                  00280000
say 'Running CSNBSYE to perform AES Encryption'                           00290000
say '        (Protected key)  '                                           00300000
say ' ';                                                                  00310000
say '         Key = '||key                                                00320000
say '        Data = '||data                                               00330000
say ' Data length = '||length(data)                                       00340000
say ' ';                                                                  00350000
                                                                          00360000
ex_rc = '00000000'X                                                       00370000
ex_rs = '00000000'X                                                       00380000
exit_data_length = '00000000'X                                            00390000
exit_data        = '        '                                             00400000
rule_array_count = '00000003'x                                            00410000
rule_array       = 'AES     KEYIDENTINITIAL '                             00420000
key_length       = '00000040'x          /* Length 64 for label */       00430000
key_identifier   = left(key,64,' ')                                       00440000
key_parms_len    = '00000000'x                                            00450000
key_parms        = '    '                                                 00460000
block_size       = '00000010'x                                            00470000
init_vector_len  = '00000010'x                                            00480000
init_vector      = '00000000000000000000000000000000'X                    00490000
```

```
chain_data_len   = '00000020'x                                          00500000
chain_data       = COPIES('00'X,32)                                     00510000
clear_text_len   = '00000030'x                                          00520000
clear_text       = left(data,48,'00'X)                                  00530000
cipher_text_len  = '00000030'x                                          00540000
cipher_text      = COPIES('00'X,48)                                     00550000
opt_data_len     = '00000000'x                                          00560000
opt_data         = '00000000'X                                          00570000
                                                                        00580000
address linkpgm 'CSNBSYE',                                             00590000
                'ex_rc',                                               00600000
                'ex_rs',                                               00610000
                'exit_data_length',                                    00620000
                'exit_data',                                           00630000
                'rule_array_count',                                    00640000
                'rule_array',                                          00650000
                'key_length',                                          00660000
                'key_identifier',                                      00670000
                'key_parms_len',                                       00680000
                'key_parms',                                           00690000
                'block_size',                                          00700000
                'init_vector_len',                                     00710000
                'init_vector',                                         00720000
                'chain_data_len',                                      00730000
                'chain_data',                                          00740000
                'clear_text_len',                                      00750000
                'clear_text',                                          00760000
                'cipher_text_len',                                     00770000
                'cipher_text',                                         00780000
                'opt_data_len',                                        00790000
                'opt_data'                                             00800000
/* check the return code */                                            00810000
                                                                       00820000
if (ex_rc <> '00000000'X) | (ex_rs <> '00000000'X) then do             00830000
  say 'CSNBSYE FAILed : RC =' c2x(ex_rc);                              00840000
  say '                 RS =' c2x(ex_rs);                              00850000
  say ' ' ;                                                            00860000
  end ;                                                                00870000
else do ;                                                              00880000
  say 'CSNBSYE OK '                                                    00890000
  say 'Clear_Text  (part1)...'||c2x(SUBSTR(clear_text,01,16))          00900000
  say 'Cipher_Text (part1)...'||c2x(SUBSTR(cipher_text,01,16))         00910000
  say ' ' ;                                                            00920000
  say 'Clear_Text  (part2)...'||c2x(SUBSTR(clear_text,17,16))          00930000
  say 'Cipher_Text (part2)...'||c2x(SUBSTR(cipher_text,17,16))         00940000
  say ' ' ;                                                            00950000
  say 'Clear_Text  (part3)...'||c2x(SUBSTR(clear_text,33,16))          00960000
  say 'Cipher_Text (part3)...'||c2x(SUBSTR(cipher_text,33,16))         00970000
  say ' ' ;                                                            00980000
  say 'CSNBSYE completed'                                              00990000
  end ;                                                                01000000
                                                                       01010000
/*--------------------------------------------------------------------*/ 01020000
/*                                                                    */ 01030000
/*   Invoke CSNBSAD                                                   */ 01040000
```

```
/*                                                                  */  01050000
/*------------------------------------------------------------------*/  01060000
                                                                        01070000
/* ************************************************** */            01080000
/* initialize parameters for common use               */            01090000
/*                                                    */            01100000
                                                                        01110000
/* trace i                                            */            01120000
say ' ';                                                            01130000
say ' ';                                                            01140000
say ' ';                                                            01150000
say ' ';                                                            01160000
say '-----------------------------------------';                    01170000
say ' ';                                                            01180000
say 'Running CSNBSAD to perform AES Decryption'                     01190000
say '        (secure KEY)  '                                        01200000
say ' ';                                                            01210000
/*  ex_rc            = (from above)     */                          01220000
/*  ex_rs            = (from above)     */                          01230000
/*  exit_data_length = (from above)     */                          01240000
/*  exit_data        = (from above)     */                          01250000
/*  rule_array_count = (from above)     */                          01260000
/*  rule_array       = (from above)     */                          01270000
/*  key_length       = (from above)     */                          01280000
/*  key_identifier   = (from above)     */                          01290000
/*  key_parms_len    = (from above)     */                          01300000
/*  key_parms        = (from above)     */                          01310000
/*  block_size       = (from above)     */                          01320000
/*  init_vector_len  = (from above)     */                          01330000
/*  init_vector      = (from above)     */                          01340000
/*  chain_data_len   = (from above)     */                          01350000
/*  chain_data       = (from above)     */                          01360000
clear_text_len   = '00000030'x                                      01370000
clear_text       = left(data,48,'00'X)                              01380000
/*  cipher_text_len  = (from above)     */                          01390000
/*  cipher_text      = (from above)     */                          01400000
/*  opt_data_len     = (from above)     */                          01410000
/*  opt_data         = (from above)     */                          01420000
                                                                    01430000
address linkpgm 'CSNBSAD',                                          01440000
                'ex_rc',                                            01450000
                'ex_rs',                                            01460000
                'exit_data_length',                                 01470000
                'exit_data',                                        01480000
                'rule_array_count',                                 01490000
                'rule_array',                                       01500000
                'key_length',                                       01510000
                'key_identifier',                                   01520000
                'key_parms_len',                                    01530000
                'key_parms',                                        01540000
                'block_size',                                       01550000
                'init_vector_len',                                  01560000
                'init_vector',                                      01570000
                'chain_data_len',                                   01580000
                'chain_data',                                       01590000
```

```
                'cipher_text_len',                                      01600000
                'cipher_text',                                          01610000
                'clear_text_len',                                       01620000
                'clear_text',                                           01630000
                'opt_data_len',                                         01640000
                'opt_data'                                              01650000
/* check the return code */                                             01660000
                                                                        01670000
if (ex_rc <> '00000000'X) | (ex_rs <> '00000000'X) then do              01680000
  say 'CSNBSAD FAILed : RC =' c2x(ex_rc);                               01690000
  say '                 RS =' c2x(ex_rs);                               01700000
  say ' ' ;                                                             01710000
  end ;                                                                 01720000
else do ;                                                               01730000
  say 'CSNBSAD OK '                                                     01740000
  say 'Cipher_Text (part1)...'||c2x(SUBSTR(cipher_text,01,16))          01750000
  say 'Clear_Text  (part1)...'||c2x(SUBSTR(clear_text,01,16))           01760000
  say ' ' ;                                                             01770000
  say 'Cipher_Text (part2)...'||c2x(SUBSTR(cipher_text,17,16))          01780000
  say 'Clear_Text  (part2)...'||c2x(SUBSTR(clear_text,17,16))           01790000
  say ' ' ;                                                             01800000
  say 'Cipher_Text (part3)...'||c2x(SUBSTR(cipher_text,33,16))          01810000
  say 'Clear_Text  (part3)...'||c2x(SUBSTR(clear_text,33,16))           01820000
  say ' ' ;                                                             01830000
  say 'Clear_Text = 'clear_text                                         01840000
  Say 'Clear_Text length = '||length(clear_text)                        01850000
  say ' ' ;                                                             01860000
  say 'CSNBSAD completed'                                               01870000
  end ;                                                                 01880000
                                                                        01890000
if data == clear_text then do                                           01900000
  say ' '                                                               01910000
  say 'Data decrypted by CSNBSAD using clear key, exactly matches'      01920000
  say 'original data encrypted by CSNBSYE using protected key.'         01930000
  end                                                                   01940000
exit                                                                    01950000
```

CPACF040 assembler program

This program is an assembler subroutine to read a CLEAR key and then make it available as a protected key.

The example JCL that we use makes use of a clear AES key, but this program can also be used for clear DES keys. The reader is left to make use of it for DES encryption, as an exercise.

```
CPACF040 CSECT                                                          00010000
CPACF040 AMODE 31                                                       00020000
CPACF040 RMODE ANY                                                      00030000
*                                                                       00040000
         MACRO                                                          00050000
&LAB     PCKMO ,                                                        00060000
         DC    XL4'B9280000'                                            00070000
         MEND                                                           00080000
*                                                                       00090000
* ---------------------------------------------------------------- * 00100000
*                                                                  * 00110000
*      C P A C F 0 4 0   -  Convert clear key to protected         * 00120000
*      ------------------------------------------------            * 00130000
*                                                                  * 00140000
*        Last Updated 18/05/2010                                   * 00150000
*                                                                  * 00160000
*         Module : CPACF040                                        * 00170000
*   Date written : 18th May 2010                                   * 00180000
*       Function : Convert a CLEAR key to PROTECTED key            * 00190000
*                                                                  * 00200000
*        HISTORY : 18/05/2010 - Initial Version                    * 00210000
*                                                                  * 00220000
*   Module attributes: RENT,REUS,AMODE(31),RMODE(31),AC(1)         * 00230000
*                                                                  * 00240000
* ---------------------------------------------------------------- * 00250000
*                                                                    00260000
* ---------------------------------------------------------------- * 00270000
*                                                                  * 00280000
*  Function: Read a CLEAR key from the CKDS using CSNBKRR, and     * 00290000
*            the key from it. Pass the key to the PCKMO instruction* 00300000
*            to create a Protected Key version, with a corresponding* 00310000
*            verification pattern.                                 * 00320000
*                                                                  * 00330000
*        Parameters: 1 - Return Code                 4 bytes R7    * 00340000
*                    2 - Reason Code                 4 bytes R8    * 00350000
*                    3 - Key label                  64 bytes R9    * 00360000
*                    4 - Wrapped Key                64 bytes R10   * 00370000
*                        and verification pattern                  * 00380000
*                                                                  * 00390000
*      Return Codes: 00 - All OK                                   * 00400000
*                    04 - ICSF return code                         * 00410000
*                    08 - ICSF return code                         * 00420000
*                    12 - ICSF return code                         * 00430000
*                    16 - ICSF return code                         * 00440000
*                    20 - Not a clear key                          * 00450000
*                    24 - Failure to read record                   * 00460000
```

```
*                   28 - AES token has bad length                   * 00470000
*                   32 - Bad Token Version                          * 00480000
*                                                                   * 00490000
* ----------------------------------------------------------------- * 00500000
         SAVE  (14,12),,CPACFO4O-&SYSDATE-&SYSTIME                    00510000
*                                                                    00520000
         LR    R12,R15                                               00530000
*                                                                    00540000
         USING CPACFO4O,R12        Module addressability             00550000
*                                                                    00560000
         LR    R1O,R1              Save param address                00570000
*                                                                    00580000
         LA    RO,WkEnd-WkStart    Length of work area               00590000
         XR    R15,R15             Subpool 0                         00600000
*                                                                    00610000
         STORAGE OBTAIN,SP=(15),ADDR=(1),LENGTH=(0)                   00620000
*                                                                    00630000
         USING #Work,R1            Workarea addressability           00640000
*                                                                    00650000
         ST    RO,WkLen            Save length                       00660000
         ST    R15,WkSpl           Save subpool                      00670000
*                                                                    00680000
MainOOO  DS    OH                                                    00690000
         ST    R1,8(,R13)          Link save areas                   00700000
         ST    R13,4(,R1)              for os                        00710000
*                                                                    00720000
         LR    R13,R1              Set work/save area address        00730000
         LR    R1,R1O              Reinstate parameter register      00740000
*                                                                    00750000
         DROP  R1                  #Work                             00760000
         USING #Work,R13           Workarea addressability           00770000
*                                                                    00780000
         LM    R7,R1O,O(R1)        R7 to R1O point to my parameters  00790000
         STM   R7,R1O,WkParms      Store parms for debugging         00800000
         XC    WkRetcde,WkRetcde   Clear R15 return code             00810000
         XC    WkReturn,WkReturn   Clear return code                 00820000
         XC    WkReason,WkReason   Clear reason code                 00830000
         XC    WkFlags,Wkflags     Clear logic flags                 00840000
*                                                                    00850000
         BAS   R14,AuthOOO         Check auth, switch to sup state   00860000
*                                                                    00870000
         BAS   R14,ReadOOO         Read CKDS record                  00880000
*                                                                    00890000
         BAS   R14,ConvOOO         Do Encrypt processing             00900000
*                                                                    00910000
         BAS   R14,NautOOO         Lose authorisation                00920000
*                                                                    00930000
         WTO   'CPACFO4O: Terminating',ROUTCDE=11                     00940000
*                                                                    00950000
         MVC   O(4,R7),WkReturn    Store Return into callers parmlist 00960000
         MVC   O(4,R8),WkReason    Store Reason into callers parmlist 00970000
*                                                                    00980000
Main9OO  DS    OH                                                    00990000
*                                                                    01000000
         L     R1O,WkRetcde        Save return code                  01010000
```

```
         L     R0,WkLen            Length work area                    01020000
         L     R15,WkSpl           Subpool                             01030000
         LR    R1,R13              Point to my save area               01040000
         L     R13,4(R1)           Rescue pointer to high save area    01050000
*                                                                      01060000
         STORAGE RELEASE,SP=(15),ADDR=(1),LENGTH=(0)                  01070000
*                                                                      01080000
         LR    R15,R10             Reinstate return code               01090000
         RETURN (14,12),RC=(15)    Return                              01100000
*                                                                      01110000
* ------------------------------------------------------------------ * 01120000
*        Auth: Check auth and move to Sup State                      * 01130000
* ------------------------------------------------------------------ * 01140000
Auth     DC    CL4'Auth'                                             01150000
Auth000  DS    0H                                                    01160000
         STM   R0,R15,WkSaveA      Store registers                     01170000
         MVC   WkPhase,Auth        Set phase name                      01180000
*                                                                      01190000
         WTO   'CPACF040: Testing Authorisation',ROUTCDE=11          01200000
*                                                                      01210000
         TESTAUTH                                                     01220000
*                                                                      01230000
         LTR   R15,R15             Are we Auth?                        01240000
         BNZ   Auth900             No, branch                          01250000
*                                                                      01260000
         OI    WkFlag1,WkAuth      Show auth                           01270000
*                                                                      01280000
         WTO   'CPACF040: Program is APF auth',ROUTCDE=11             01290000
*                                                                      01300000
         MODESET MODE=SUP          Move to Supervisor state            01310000
*                                                                      01320000
         OI    WkFlag1,WkSup       Show in sup state                   01330000
*                                                                      01340000
Auth900  DS    0H                                                    01350000
         LM    R0,R15,WkSaveA      Reload registers                    01360000
         BR    R14                 Return to caller                    01370000
*                                                                      01380000
* ------------------------------------------------------------------ * 01390000
*        Read: Read CKDS token into storage                          * 01400000
* ------------------------------------------------------------------ * 01410000
Read     DC    CL4'Read'                                             01420000
Read000  DS    0H                                                    01430000
         STM   R0,R15,WkSaveA      Store registers                     01440000
         MVC   WkPhase,Read        Set phase name                      01450000
*                                                                      01460000
         TM    WkFlag1,WkSup       Are auth?                           01470000
         BNO   Read900             No, exit quickly                    01480000
*                                                                      01490000
         WTO   'CPACF040: Reading ICSF key',ROUTCDE=11                01500000
*                                                                      01510000
         MVC   WkLabel,0(R9)       Move Label                          01520000
         XC    WkToken,WkToken     Clear token                         01530000
         XC    WkKRRRet,WkKRRret   Clear return code                   01540000
         XC    WkKRRRea,WkKRRrea   Clear reason code                   01550000
         XC    WkKRRdal,WkKRRdal   Clear data length                   01560000
```

```
         XC    WkKRRdat,WkKRRdat   Clear data                          01570000
*                                                                       01580000
         CALL  CSNBKRR,            Read record                       C01590000
               (WkKrrRet,                                            C01600000
               WkKRRrea,                                             C01610000
               WkKRRdaL,                                             C01620000
               WkKRRdat,                                             C01630000
               WkLabel,                                              C01640000
               WkToken),                                             C01650000
               MF=(E,WKkrr),VL                                        01660000
*                                                                       01670000
         CLC   WkKrrRet,=XL4'00000000' Label read OK?                 01680000
         BNE   Read100             No, branch                         01690000
*                                                                       01700000
         CLC   WkKrrRea,=XL4'00000000' Label read OK?                 01710000
         BE    Read200             Yes, branch                        01720000
*                                                                       01730000
Read100  DS    0H                                                     01740000
         OI    WkFlag2,WkBadRd     Show record read failed            01750000
         MVC   WkRetcde,=A(BadRead)  Load return code                 01760000
         MVC   WkReturn,WkKrrRet   Store return code                  01770000
         MVC   WkReason,WkKrrRea   Store reason code                  01780000
         B     Read900                                                01790000
*                                                                       01800000
Read200  DS    0H                                                     01810000
*                                                                       01820000
TokMap   USING TOKPREF,WkToken     Map the Token prefix               01830000
AESMap   USING TOKAES,WkToken+TOKPREFL  Map AES token                  01840000
DESMap   USING TOKDES,WkToken+TOKPREFL  Map AES token                  01850000
*                                                                       01860000
         TM    TokMap.TokFlag,TokEnc Encrypted Token?                 01870000
         BNO   Read900             No, good, branch                   01880000
*                                                                       01890000
Read300  DS    0H                                                     01900000
         OI    WkFlag2,WkEncTok    Show Token encrypted               01910000
         MVC   WkRetCde,=A(NotClear)  Load return code                01920000
*        B     Read900             Branch                             01930000
*                                                                       01940000
Read900  DS    0H                                                     01950000
         LM    R0,R15,WkSaveA      Reload registers                   01960000
         BR    R14                 Return to caller                   01970000
*                                                                       01980000
* ------------------------------------------------------------------ * 01990000
*        Conv: Convert token to Protected key                        * 02000000
* ------------------------------------------------------------------ * 02010000
Conv     DC    CL4'Conv'                                              02020000
Conv000  DS    0H                                                     02030000
         STM   R0,R15,WkSaveA      Store registers                    02040000
         MVC   WkPhase,Conv        Set phase name                     02050000
         XC    WkPCKMOb,WkPCKMOb   Clear PCKMO block                  02060000
*                                                                       02070000
         TM    WkFlag1,WkSup       Are auth?                          02080000
         BNO   Conv900             No, exit quickly                   02090000
*                                                                       02100000
         WTO   'CPACF040: Converting key',ROUTCDE=11                  02110000
```

```
*                                                                       02120000
         CLI   WkFlag2,X'00'       Any errors?                          02130000
         BNE   Conv900             Yes, terminate                       02140000
*                                                                       02150000
         CLI   TokMap.TokVer,X'00' Des Token?                           02160000
         BE    Conv300             Yes, branch                          02170000
*                                                                       02180000
         CLI   TokMap.TokVer,X'01' Des Token?                           02190000
         BE    Conv300             Yes, branch                          02200000
*                                                                       02210000
         CLI   TokMap.TokVer,X'04' AES Token?                           02220000
         BE    Conv400             Yes, Branch                          02230000
*                                                                       02240000
         MVC   WkRetcde,=A(BadTokV)  Set return code                    02250000
         B     Conv900             Yes, continue                        02260000
*                                                                       02270000
Conv300  DS    0H                  **** DES TOKEN ****                  02280000
         TM    DESMap.TokDFlag,TokLen24 Is this triple length?          02290000
         BO    Conv340             Yes, Branch                          02300000
*                                                                       02310000
         TM    DESMap.TokDFlag,TokLen16 Is this double length?          02320000
         BO    Conv320             Yes, Branch                          02330000
*                                                                       02340000
         MVC   WkPCKMOb(8),DESMap.TokDesK1 Move in key                  02350000
         XC    DESMap.TokDESK1(8),DESMap.TokDESK1  Clear storage        02360000
         LA    R3,WkPCKMOb+8       Point at Verification pattern        02370000
         LA    R0,1                Load Function code                   02380000
         B     Conv700             Go do PCKMO                          02390000
*                                                                       02400000
Conv320  DS    0H                                                       02410000
         MVC   WkPCKMOb+00(8),DESMap.TokDesK1 Move in key               02420000
         MVC   WkPCKMOb+08(8),DESMap.TokDesk2 Move in key               02430000
         XC    DESMap.TokDESK1(8),DESMap.TokDESK1  Clear storage        02440000
         XC    DESMap.TokDESK2(8),DESMap.TokDESK2  Clear storage        02450000
         LA    R3,WkPCKMOb+16      Point at Verification pattern        02460000
         LA    R0,2                Load Function code                   02470000
         B     Conv700             Go do PCKMO                          02480000
*                                                                       02490000
Conv340  DS    0H                                                       02500000
         MVC   WkPCKMOb+00(8),DESMap.TokDesK1 Move in key               02510000
         MVC   WkPCKMOb+08(8),DESMap.TokDesk2 Move in key               02520000
         MVC   WkPCKMOb+16(8),DESMap.TokDesk3 Move in key               02530000
         XC    DESMap.TokDESK1(8),DESMap.TokDESK1  Clear storage        02540000
         XC    DESMap.TokDESK2(8),DESMap.TokDESK2  Clear storage        02550000
         XC    DESMap.TokDESK3(8),DESMap.TokDESK3  Clear storage        02560000
         LA    R3,WkPCKMOb+24      Point at Verification pattern        02570000
         LA    R0,3                Load Function code                   02580000
         B     Conv700             Go do PCKMO                          02590000
*                                                                       02600000
Conv400  DS    0H                                                       02610000
         CLC   AESMap.TokAESLn,=H'128' 128 bit key?                     02620000
         BE    Conv420             Yes, branch                          02630000
*                                                                       02640000
         CLC   AESMap.TokAESLn,=H'192' 192 bit key?                     02650000
         BE    Conv440                                                  02660000
```

```
*                                                                       02670000
         CLC   AESMap.TokAESLn,=H'256' 256 bit key?                     02680000
         BE    Conv460                                                  02690000
*                                                                       02700000
         MVC   WkRetcde,=A(BADAESLN)  Show bad AES key length           02710000
         B     Conv900             Exit                                 02720000
*                                                                       02730000
Conv420  DS    0H                                                       02740000
         MVC   WkPCKMOb(16),AESMap.TokAESKy Move 16 byte key            02750000
         XC    AESMap.TokAESKy(16),AESMap.TokAESKy  Clear storage       02760000
         LA    R3,WkPCKMOb+16      Point at Verification pattern        02770000
         LA    R0,18               Load Function code                   02780000
         B     Conv700                                                  02790000
*                                                                       02800000
Conv440  DS    0H                                                       02810000
         MVC   WkPCKMOb(24),AESMap.TokAESKy Move 24 byte key            02820000
         XC    AESMap.TokAESKy(24),AESMap.TokAESKy  Clear storage       02830000
         LA    R3,WkPCKMOb+24      Point at Verification pattern        02840000
         LA    R0,19               Load Function code                   02850000
         B     Conv700                                                  02860000
*                                                                       02870000
Conv460  DS    0H                                                       02880000
         MVC   WkPCKMOb(32),AESMap.TokAESKy Move 32 byte key            02890000
         XC    AESMap.TokAESKy(32),AESMap.TokAESKy  Clear storage       02900000
         LA    R3,WkPCKMOb+32      Point at Verification pattern        02910000
         LA    R0,20               Load Function code                   02920000
*                                                                       02930000
Conv700  DS    0H                                                       02940000
         LA    R1,WkPCKMOb         Point to PCKMO block                 02950000
         PCKMO ,                   Issue PCKMO                          02960000
*                                                                       02970000
         MVC   0(64,R10),WkPCKMOb  Set return parameter                 02980000
*                                                                       02990000
Conv900  DS    0H                                                       03000000
         MVC   0(4,R7),WkRetcde    Store RC into callers parmlist       03010000
         LM    R0,R15,WkSaveA      Reload registers                     03020000
         BR    R14                 Return to caller                     03030000
*                                                                       03040000
* ------------------------------------------------------------------ * 03050000
*        Naut: Revert to problem state                                * 03060000
* ------------------------------------------------------------------ * 03070000
Naut     DC    CL4'Naut'                                                03080000
Naut000  DS    0H                                                       03090000
         STM   R0,R15,WkSaveA      Store registers                      03100000
         MVC   WkPhase,Naut        Set phase name                       03110000
*                                                                       03120000
         TM    WkFlag1,WkSup       Are we in Supervisor state?          03130000
         BNO   Naut900             NO, branch                           03140000
*                                                                       03150000
         WTO   'CPACF040: Losing Authorisation',ROUTCDE=11              03160000
*                                                                       03170000
         MODESET MODE=PROB         Move back to problem state           03180000
*                                                                       03190000
         NI    WkFlag1,X'FF'-WkSup Show no Sup state                    03200000
*                                                                       03210000
```

```
Naut900  DS    0H                                                      03220000
         LM    R0,R15,WkSaveA      Reload registers                    03230000
         BR    R14                 Return to caller                    03240000
*                                                                      03250000
         DROP  TokMap              Unmap Token prefix                  03260000
         DROP  AESMap              Unmap AES Token                     03270000
         DROP  DESMap              Unmap DES Token                     03280000
         DROP  R12                 CPACF040                            03290000
         DROP  R13                 #Work                               03300000
         LTORG                                                         03310000
*                                                                      03320000
         TITLE '***** CPACF040 : EQUATES *****'                        03330000
*                                                                      03340000
R0       EQU   0                                                       03350000
R1       EQU   1                                                       03360000
R2       EQU   2                                                       03370000
R3       EQU   3                                                       03380000
R4       EQU   4                                                       03390000
R5       EQU   5                                                       03400000
R6       EQU   6                                                       03410000
R7       EQU   7                                                       03420000
R8       EQU   8                                                       03430000
R9       EQU   9                                                       03440000
R10      EQU   10                                                      03450000
R11      EQU   11                                                      03460000
R12      EQU   12                                                      03470000
R13      EQU   13                                                      03480000
R14      EQU   14                                                      03490000
R15      EQU   15                                                      03500000
*                                                                      03510000
NotClear EQU   20    20 - Not a clear key                            * 03520000
BadRead  EQU   24    24 - Failure to read record                     * 03530000
BadAESLN EQU   28    28 - AES token has bad length                   * 03540000
BadTokV  EQU   32    32 - Bad Token Version                          * 03550000
*                                                                      03560000
         TITLE '***** CPACF040 : WORK AREA *****'                      03570000
#Work    DSECT                                                         03580000
WkStart  EQU   *                                                       03590000
WkOsSave DS    18F                 OS save area                       03600000
WkLen    DS    F                   Length                             03610000
WkSpl    DS    F                   Subpool                            03620000
*                                                                      03630000
WkPhase  DS    CL4                 Phase name                         03640000
WkSaveA  DS    16F                 Subroutine save area 1             03650000
*                                                                      03660000
WkParms  DS    4F                  Parm addresses                     03670000
*                                                                      03680000
WkFlags  DS    0F                  Logic flags                        03690000
WkFlag1  DS    X                   Logic flag 1                       03700000
WkAuth   EQU   B'10000000'         APF Auth                           03710000
WkSup    EQU   B'01000000'         Supervisor state                   03720000
*                                                                      03730000
WkFlag2  DS    X                   Error flags                        03740000
WkBadRd  EQU   B'10000000'         Read unsuccessful                  03750000
WkEncTok EQU   B'01000000'         Token is for an encrypted key      03760000
```

```
*                                                                       03770000
WkFlag3  DS    X                 Logic flag 3                         03780000
*                                                                       03790000
WkFlag4  DS    X                 Logic flag 4                         03800000
*                                                                       03810000
WkRetcde DS    F                 R15 on exit                          03820000
WkReturn DS    F                 Return Code                          03830000
WkReason DS    F                 Reason Code                          03840000
*                                                                       03850000
WkPCKMOb DS    XL64              Max length of PCKMO parm block       03860000
*                                                                       03870000
WkKrr    DS    6A                CSNBKRR parmlist                     03880000
         ORG   WkKrr                                                   03890000
WkKRARet DS    A                 ->  Return Code                      03900000
WkKRARea DS    A                 ->  Reason Code                      03910000
WkKRADal DS    A                 ->  Data Length                      03920000
WkKRADat DS    A                 ->  Data                             03930000
WkKRALab DS    A                 ->  Label                            03940000
WkKRATok DS    A                 ->  Token                            03950000
         ORG                                                           03960000
*                                                                       03970000
WkKrrRet DS    F                 Return Code                          03980000
WkKrrRea DS    F                 Reason Code                          03990000
WkKRRDal DS    F                 Data Length                          04000000
WkKRRDat DS    A                 Dummy Data                           04010000
WkToken  DS    XL64              Token                                04020000
WkLabel  DS    CL64              Label                                04030000
*                                                                       04040000
WkEnd    EQU   *                                                      04050000
*                                                                       04060000
TOKPREF  DSECT                                                        04070000
         DS    X                                                      04080000
         DS    3X                                                     04090000
TOKVer   DS    X                 X'00',X'01'=DES, X'04'=AES           04100000
         DS    X                                                      04110000
TokFlag  DS    X                 Flag byte                            04120000
TokEnc   EQU   B'10000000'       Bit 1, On=Enc, Off=Clear             04130000
         DS    X                                                      04140000
         DS    8X                MKVP                                 04150000
TOKPREFL EQU   *-TOKPREF                                              04160000
*                                                                       04170000
TOKAES   DSECT                                                        04180000
TokAESKY DS    XL32              AES key                              04190000
         DS    8X                                                     04200000
TokAESLN DS    2X                Length AES key (in bits)             04210000
         DS    2X                                                     04220000
         DS    4X                                                     04230000
*                                                                       04240000
TOKDES   DSECT                                                        04250000
TokDesK1 DS    8X                Des Key Part 1                       04260000
TokDesK2 DS    8X                Des Key Part 2                       04270000
TokDesCV DS    8X                                                     04280000
         DS    8X                                                     04290000
TokDesK3 DS    8X                Des Key Part 3                       04300000
         DS    3X                                                     04310000
```

```
TokDflag DS    X                                                        04320000
TokLen08 EQU   B'00000000'                                              04330000
TokLen16 EQU   B'00010000'                                              04340000
TokLen24 EQU   B'00100000'                                              04350000
         DS    4X                TVV                                   04360000
*                                                                       04370000
         END                                                            04380000
```

CPACF050 assembler program

This program is an assembler subroutine to read a file containing a clear key label and then pass it to CPACF040. The resulting wrapped key and the verification pattern are stored in an output file.

This program must be link-edited with the CPACF040 program, and the ICSF load library containing the stub routine for CSNBKRR (or CSFKRR) must be included on the SYSLIB statement. The resulting load module should be marked AC=1, and should be stored in an APF authorized library.

```
CPACF050 CSECT                                                          00010000
CPACF050 AMODE 31                                                       00020000
CPACF050 RMODE 24                                                       00030000
* ---------------------------------------------------------------- * 00040000
*                                                                  * 00050000
*      C P A C F 0 5 0  -  Invoke CPACF040 from Batch              * 00060000
*      ---------------------------------------------              * 00070000
*                                                                  * 00080000
*        Last Updated 19/05/2010                                   * 00090000
*                                                                  * 00100000
*         Module : CPACF050                                        * 00110000
*   Date written : 19th May 2010                                   * 00120000
*       Function : Invoke CPACF040                                 * 00130000
*                                                                  * 00140000
*        HISTORY : 19/05/2010 - Initial Version                    * 00150000
*                                                                  * 00160000
*   Module attributes: RENT,REUS,AMODE(31),RMODE(31),AC(1)         * 00170000
*                                                                  * 00180000
* ---------------------------------------------------------------- * 00190000
*   Function: Read a clear key label name and pass to CPACF040     * 00200000
*             which will produce a PROTECTED key version and a     * 00210000
*             verification pattern. Write the key and verification * 00220000
*             pattern to an output file.                           * 00230000
*                                                                  * 00240000
*   Files:  LABEL   LRECL=80 RECFM=FB                              * 00250000
*                   Contains a single record with the name of      * 00260000
*                   a CLEAR ICSF key in the first 64 bytes         * 00270000
*                                                                  * 00280000
*           OUTPUT  LRECL=80 RECFM=FB                              * 00290000
*                   Single 80 byte record will be written containing * 00300000
*                   the protected key and verification pattern in  * 00310000
*                   a single record. Max data used will be 64 bytes. * 00320000
*                                                                  * 00330000
* ---------------------------------------------------------------- * 00340000
         SAVE  (14,12),,CPACF050-&SYSDATE-&SYSTIME                  00350000
*                                                                   00360000
         LR    R12,R15                                              00370000
*                                                                   00380000
         USING CPACF050,R12        Module addressability            00390000
*                                                                   00400000
         LR    R10,R1              Save param address               00410000
*                                                                   00420000
         LA    R0,WkEnd-WkStart    Length of work area              00430000
         XR    R15,R15             Subpool 0                        00440000
```

```
*                                                                       00450000
         STORAGE OBTAIN,SP=(15),ADDR=(1),LENGTH=(0)                     00460000
*                                                                       00470000
         USING #Work,R1            Workarea addressability              00480000
*                                                                       00490000
         ST    R0,WkLen            Save length                          00500000
         ST    R15,WkSpl           Save subpool                         00510000
*                                                                       00520000
Main000  DS    0H                                                       00530000
         ST    R1,8(,R13)          Link save areas                      00540000
         ST    R13,4(,R1)              for os                           00550000
*                                                                       00560000
         LR    R13,R1              Set work/save area address           00570000
         LR    R1,R10              Reinstate parameter register         00580000
*                                                                       00590000
         DROP  R1                  #Work                                00600000
         USING #Work,R13           Workarea addressability              00610000
*                                                                       00620000
         XC    WkRetcde,WkRetcde   Clear R15 return code                00630000
         XC    Wkflags,Wkflags     Clear logic flags                    00640000
*                                                                       00650000
         BAS   R14,Read000         Open file read Key Label             00660000
*                                                                       00670000
         BAS   R14,Invk000         Invoke CPACF040                      00680000
*                                                                       00690000
         BAS   R14,Writ000         Write output file wrapped key        00700000
*                                                                       00710000
         WTO   'CPACF050: Terminating',ROUTCDE=11                       00720000
*                                                                       00730000
Main800  DS    0H                                                       00740000
         MVC   WkRetcde+3(1),WkFlag1  Store flags in retcode            00750000
         TM    WkFlag1,WkC40fai    CPACF040 failed ?                    00760000
         BNO   Main900             No, continue                         00770000
*                                                                       00780000
         MVC   WkRetcde(3),WkC40Ret  Move in CPACF040 return code       00790000
*                                                                       00800000
Main900  DS    0H                                                       00810000
         L     R10,WkRetcde        Save return code                     00820000
         L     R0,WkLen            Length work area                     00830000
         L     R15,WkSpl           Subpool                              00840000
         LR    R1,R13              Point to my save area                00850000
         L     R13,4(R1)           Rescue pointer to high save area     00860000
*                                                                       00870000
         STORAGE RELEASE,SP=(15),ADDR=(1),LENGTH=(0)                    00880000
*                                                                       00890000
         LR    R15,R10             Reinstate return code                00900000
         RETURN (14,12),RC=(15)    Return                               00910000
*                                                                       00920000
* ------------------------------------------------------------------ * 00930000
*        Read: Read CKDS label                                       * 00940000
* ------------------------------------------------------------------ * 00950000
Read     DC    CL4'Read'                                                00960000
Read000  DS    0H                                                       00970000
         STM   R0,R15,WkSaveA      Store registers                      00980000
         MVC   WkPhase,Read        Set phase name                       00990000
```

```
*                                                                       01000000
         WTO   'CPACF050: Opening LABEL file',ROUTCDE=11                01010000
*                                                                       01020000
         MVC   WkDCBELb(DCBELbLn),DCBELB  Format DCBE                   01030000
         MVC   WkDCBLAB(DCBLABLn),DCBLAB  Format DCB                    01040000
         LA    R3,WkDCBLAB         Point at DCB                       01050000
*                                                                       01060000
         USING IHADCB,R3                                                01070000
*                                                                       01080000
         LA    R1,WkDCBELb         Point at DCBE                      01090000
         ST    R1,DCBDCBE          Store into DCB                     01100000
         MVC   WkOpenL(OPENLEN),OPENL Format Open                       01110000
*                                                                       01120000
         OPEN  ((R3),(INPUT)),MF=(E,WkOpenL)                            01130000
*                                                                       01140000
         TM    DCBOFLGS,DCBOFOPN   Open OK?                             01150000
         BO    Read100                                                  01160000
*                                                                       01170000
         OI    WkFlag1,WkOpLabF    Show open failed                     01180000
         B     Read900                                                  01190000
*                                                                       01200000
Read100  DS    0H                                                       01210000
*                                                                       01220000
         GET   (R3)                Get 1 record                       01230000
*                                                                       01240000
         MVC   WkC40Lab,0(R1)      Get label name                       01250000
         B     Read300                                                  01260000
*                                                                       01270000
Read200  DS    0H                  *** EOF ***                        01280000
         OI    Wkflag1,WKEOF       EOF encountered                    01290000
*                                                                       01300000
Read300  DS    0H                                                       01310000
         LA    R3,WkDCBLAB         Point at DCB                       01320000
         MVC   WkCloseL(CLOSELEN),CloseL  Format Close                  01330000
*                                                                       01340000
         CLOSE ((R3)),MF=(E,WkCLOSEL),MODE=31                           01350000
*                                                                       01360000
Read900  DS    0H                                                       01370000
         LM    R0,R15,WkSaveA      Reload registers                   01380000
         BR    R14                 Return to caller                   01390000
*                                                                       01400000
* ------------------------------------------------------------------ * 01410000
*        Invk: Convert token to Protected key                        * 01420000
* ------------------------------------------------------------------ * 01430000
Invk     DC    CL4'Invk'                                                01440000
Invk000  DS    0H                                                       01450000
         STM   R0,R15,WkSaveA      Store registers                    01460000
         MVC   WkPhase,Invk        Set phase name                     01470000
         XC    WkC40Dat,WkC40Dat   Clear output area                  01480000
*                                                                       01490000
         WTO   'CPACF050: Invoking CPACF040',ROUTCDE=11                 01500000
*                                                                       01510000
         CLI   WkFlag1,X'00'       Any errors?                        01520000
         BNE   Invk900             Yes, stop                          01530000
*                                                                       01540000
```

```
         LA    R1,WkC4ORet         Point at Return code area           01550000
         ST    R1,WkA4ORet         Store into parm list                01560000
         LA    R1,WkC4ORea         Point at Reason code area           01570000
         ST    R1,WkA4ORea         Store into parm list                01580000
         LA    R1,WkC4OLab         Point at Label                      01590000
         ST    R1,WkA4OLab         Store into parm list                01600000
         LA    R1,WkC4ODat         Point at data area                  01610000
         ST    R1,WkA4ODat         Store into parm list                01620000
*                                                                      01630000
         LA    R1,WkA4O            Point at parm list                  01640000
*                                                                      01650000
         CALL  CPACFO4O            Call Conversion routine             01660000
*                                                                      01670000
         WTO   'CPACFO5O: CPACFO4O returned control',ROUTCDE=11        01680000
*                                                                      01690000
         L     R15,WkC4ORet        Get return code                     01700000
         LTR   R15,R15             OK ?                                01710000
         BZ    InVk9OO             Branch                              01720000
*                                                                      01730000
         WTO   'CPACFO5O: CPACFO4O non-zero retcode',ROUTCDE=11        01740000
*                                                                      01750000
         OI    WkFlag1,WkC4Ofai    Show CPACFO4O failed                01760000
*                                                                      01770000
Invk9OO  DS    OH                                                      01780000
         OC    WkC4ODat,WkC4ODat   is this zero?                       01790000
         BNZ   Invk99O             no, branch                          01800000
*                                                                      01810000
         WTO   'CPACFO5O: OO1: Protected key is null',ROUTCDE=11       01820000
*                                                                      01830000
Invk99O  DS    OH                                                      01840000
         LM    RO,R15,WkSaveA      Reload registers                    01850000
         BR    R14                 Return to caller                    01860000
*                                                                      01870000
* ------------------------------------------------------------------ * 01880000
*        Writ: Writer wrapped key to OUTPUT                          * 01890000
* ------------------------------------------------------------------ * 01900000
Writ     DC    CL4'Writ'                                               01910000
WritOOO  DS    OH                                                      01920000
         STM   RO,R15,WkSaveA      Store registers                     01930000
         MVC   WkPhase,Writ        Set phase name                      01940000
*                                                                      01950000
         WTO   'CPACFO5O: Writing Protected key',ROUTCDE=11            01960000
*                                                                      01970000
         CLI   WkFlag1,X'OO'       Any errors?                         01980000
         BNE   Writ9OO             Yes, stop                           01990000
*                                                                      02000000
         MVC   WkDCBEOU(DCBEOULn),DCBEOU  Format DCBE                  02010000
         MVC   WkDCBOUT(DCBOUTLn),DCBOUT  Format DCB                   02020000
         LA    R3,WkDCBOUT         Point at DCB                        02030000
*                                                                      02040000
         USING IHADCB,R3                                               02050000
*                                                                      02060000
         LA    R1,WkDCBEOU         Point at DCBE                       02070000
         ST    R1,DCBDCBE          Store into DCB                      02080000
         MVC   WkOpenL(OPENLEN),OPENL Format Open                      02090000
```

```
*                                                                       02100000
         OPEN  ((R3),(OUTPUT)),MF=(E,WkOpenL)                           02110000
*                                                                       02120000
         TM    DCBOFLGS,DCBOFOPN   Open OK?                             02130000
         BO    Writ100                                                  02140000
*                                                                       02150000
         OI    WkFlag1,WkOpOutF    Show open failed                     02160000
         B     Writ900                                                  02170000
*                                                                       02180000
         OC    WkC40Dat,WkC40Dat   is this zero?                        02190000
         BNZ   Invk900             no, branch                           02200000
*                                                                       02210000
         WTO   'CPACF050: 002: Protected key is null',ROUTCDE=11        02220000
*                                                                       02230000
Writ100  DS    0H                                                       02240000
*                                                                       02250000
         PUT   (R3),WkC40Dat       Put 1 record                         02260000
*                                                                       02270000
Writ900  DS    0H                                                       02280000
         LM    R0,R15,WkSaveA      Reload registers                     02290000
         BR    R14                 Return to caller                     02300000
*                                                                       02310000
         DROP  R12                 CPACF050                             02320000
         DROP  R13                 #Work                                02330000
*                                                                       02340000
         TITLE '***** CPACF050 : Data Areas *****'                      02350000
*                                                                       02360000
OpenL    OPEN  (*-*),MODE=31,MF=L                                       02370000
OpenLen  EQU   *-OpenL                                                  02380000
CloseL   CLOSE *-*,MF=L                                                 02390000
CloseLen EQU   *-CloseL                                                 02400000
DCBLAB   DCB   MACRF=(GL),DDNAME=LABEL,DSORG=PS,LRECL=80                02410000
DCBLABLn EQU   *-DCBLAB                                                 02420000
DCBOUT   DCB   MACRF=(PM),DDNAME=OUTPUT,DSORG=PS,LRECL=80               02430000
DCBOUTLn EQU   *-DCBOUT                                                 02440000
DCBELB   DCBE  EODAD=Read200,RMODE31=BUFF                               02450000
DCBELBLn EQU   *-DCBELB                                                 02460000
DCBEOU   DCBE  RMODE31=BUFF                                             02470000
DCBEOULn EQU   *-DCBEOU                                                 02480000
*                                                                       02490000
         LTORG                                                          02500000
         TITLE '***** CPACF050 : EQUATES *****'                         02510000
*                                                                       02520000
R0       EQU   0                                                        02530000
R1       EQU   1                                                        02540000
R2       EQU   2                                                        02550000
R3       EQU   3                                                        02560000
R4       EQU   4                                                        02570000
R5       EQU   5                                                        02580000
R6       EQU   6                                                        02590000
R7       EQU   7                                                        02600000
R8       EQU   8                                                        02610000
R9       EQU   9                                                        02620000
R10      EQU   10                                                       02630000
R11      EQU   11                                                       02640000
```

```
R12      EQU   12                                                      02650000
R13      EQU   13                                                      02660000
R14      EQU   14                                                      02670000
R15      EQU   15                                                      02680000
*                                                                      02690000
         TITLE '***** CPACF05O : WORK AREA *****'                      02700000
#Work    DSECT                                                         02710000
WkStart  EQU   *                                                       02720000
WkOsSave DS    18F                OS save area                        02730000
WkLen    DS    F                  Length                              02740000
WkSpl    DS    F                  Subpool                             02750000
*                                                                      02760000
WkPhase  DS    CL4                Phase name                          02770000
WkSaveA  DS    16F                Subroutine save area 1              02780000
*                                                                      02790000
WkFlags  DS    OF                 Logic flags                         02800000
*                                                                      02810000
WkFlag1  DS    X                  Logic flag 1                        02820000
WkOpLabF EQU   B'1OOOOOOO'        Open Failed                         02830000
WkEOF    EQU   B'O1OOOOOO'        Unexpected EOF                      02840000
WkOpOutF EQU   B'OO1OOOOO'        Open failed on output               02850000
WkC4OfaiEQU   B'OOO1OOOO'        CPACF040 failed                     02860000
*                                                                      02870000
WkFlag2  DS    X                  Logic flag 2                        02880000
WkFlag3  DS    X                  Logic flag 3                        02890000
WkFlag4  DS    X                  Logic flag 4                        02900000
*                                                                      02910000
WkOpenL  OPEN  (*-*),MODE=31,MF=L                                      02920000
WKCloseL CLOSE *-*,MF=L                                                02930000
WkDCBLAB DCB   MACRF=(GL),DDNAME=LABEL,DSORG=PS                        02940000
WkDCBOUT DCB   MACRF=(PM),DDNAME=OUTPUT,DSORG=PS                       02950000
WkDCBELB DCBE  EODAD=*-*,RMODE31=BUFF                                  02960000
WkDCBEOU DCBE  EODAD=*-*,RMODE31=BUFF                                  02970000
*                                                                      02980000
WkRetcde DS    F                  R15 on exit                         02990000
*                                                                      03000000
WkA4O    DS    OA                 CPACF040 parm list                  03010000
WkA4ORet DS    A                  CPACF040 return code                03020000
WkA4ORea DS    A                  CPACF040 reason code                03030000
WkA4OLab DS    A                  CPACF040 label                      03040000
WkA4ODat DS    A                  CPACF040 token * VP                 03050000
*                                                                      03060000
WkC4ORet DS    F                                                      03070000
WkC4ORea DS    F                                                      03080000
WkC4OLab DS    XL64               Label                               03090000
WkC4ODat DS    XL8O               Wrapped key and VP                  03100000
*                                                                      03110000
WkEnd    EQU   *                                                       03120000
*                                                                      03130000
         DCBD  DSORG=PS                                                03140000
*                                                                      03150000
         END                                                           03160000
```

CPACF001 JCL

This is the JCL that we used to invoke CPACF050. Before running this JCL you must create two 80-byte record files. The first will have the label of the CLEAR AES key in bytes 1 - 44. The second should be empty. After running this JCL it will contain the AES key in wrapped format, with the verification pattern.

Note that the code supports a DES key, but we tested with AES keys. The DES results are left as an exercise for the reader.

```
//LENNIEP JOB   (999,POK),CLASS=A,MSGCLASS=A,MSGLEVEL=(1,1),
//         NOTIFY=LENNIE,REGION=4M
//*
//*  Make a CLEAR AES or DES key available as a Protected key
//*  The Wrapped key and the Verification pattern are returned.
//*
//CPACF050 EXEC  PGM=CPACF050
//STEPLIB   DD   DISP=SHR,DSN=LENNIE.A.LOAD   <--- APF auth required
//*
//* The LABEL file contains the label of a CLEAR key in the
//* first 64 bytes of the first record
//*
//LABEL     DD   DISP=SHR,DSN=LENNIE.LABEL.KEY
/*
//*
//* The OUTPUT file will contain the protected key and the
//* verification pattern in the first 64 bytes.
//* NOTE: 64 bytes is maximum needed, may be smaller.
//*
//OUTPUT    DD   DISP=SHR,DSN=LENNIE.PROTECT.KEY
//*
//* SYSUDUMP: For pessimists only
//*
//SYSUDUMP  DD   SYSOUT=*
```

CPACF010 assembler program

This is the CPACF010 program that is used to encrypt data using protected keys. This program also performs clear key encryption using CPACF, and perform DES encryption as well as AES encryption. It is an updated version of the program used in *z9-109 Crypto and TKE V5 Update*, SG24-7123, to perform clear key encryption.

```
CPACF010 CSECT                                                          00010036
CPACF010 AMODE 31                                                       00020036
CPACF010 RMODE ANY                                                      00030036
* ---------------------------------------------------------------- * 00040036
*                                                                  * 00050036
*      C P A C F 0 1 0   -  Encrypt test block using KMC           * 00060036
*      ---------------------------------------------              * 00070036
*                                                                  * 00080036
*        Last Updated 17/05/2010                                   * 00090038
*                                                                  * 00100036
*         Module : CPACF010                                        * 00110036
*   Date written : 29th August 2005                                * 00120036
*       Function : Encrypt a text block                            * 00130036
*                                                                  * 00140036
*        HISTORY : 29/08/2005 - Initial Version                    * 00150036
*                  17/05/2010 - Add Protected Key support          * 00160038
*                                                                  * 00170036
*   Module attributes: RENT,REUS,AMODE(31),RMODE(31)               * 00180036
*                                                                  * 00190036
* ---------------------------------------------------------------- * 00200036
*                                                                    00210036
* ---------------------------------------------------------------- * 00220036
*                                                                  * 00230036
*        Parameters: 1 - Return Code                               * 00240036
*                    2 - Option Block                              * 00250036
*                    3 - Data to Encrypt or Decrypt  (INPUT)       * 00260036
*                    4 - Encrypted or Decrypted Data (OUTPUT)      * 00270036
*                    5 - Key (with verification pattern if required)  * 00280038
*                                                                  * 00290036
*        Option Block format                                       * 00300036
*              Byte 1-1 Function Code                              * 00310036
*              Byte 2-2 E = Encrypt, D = Decrypt                   * 00320036
*              Byte 3-4 Length of Data                             * 00330036
*                                                                  * 00340036
*                                                                  * 00350036
* ---------------------------------------------------------------- * 00360036
         SAVE  (14,12),,CPACF010-&SYSDATE-&SYSTIME                  00370036
*                                                                   00380036
         LR    R12,R15                                              00390036
*                                                                   00400036
         USING CPACF010,R12        Module addressability            00410036
*                                                                   00420036
         LR    R10,R1              Save param address               00430036
*                                                                   00440036
         LA    R0,WkEnd-WkStart    Length of work area              00450036
         XR    R15,R15             Subpool 0                        00460036
*                                                                   00470036
         STORAGE OBTAIN,SP=(15),ADDR=(1),LENGTH=(0)                 00480036
```

```
*                                                                       00490036
         USING #Work,R1            Workarea addressability             00500036
*                                                                       00510036
         ST    R0,WkLen            Save length                         00520036
         ST    R15,WkSpl           Save subpool                        00530036
*                                                                       00540036
Main000  DS    0H                                                      00550036
         ST    R1,8(,R13)          Link save areas                     00560036
         ST    R13,4(,R1)              for os                          00570036
*                                                                       00580036
         LR    R13,R1              Set work/save area address          00590036
         LR    R1,R10              Reinstate parameter register        00600036
*                                                                       00610036
         DROP  R1                  #Work                               00620036
         USING #Work,R13           Workarea addressability             00630036
*                                                                       00640036
         BAS   R14,Encr000         Do Encrypt processing               00650036
*                                                                       00660036
Main900  DS    0H                                                      00670036
*                                                                       00680036
         L     R10,WkRetcde        Save return code                    00690036
         L     R0,WkLen            Length work area                    00700036
         L     R15,WkSpl           Subpool                             00710036
         LR    R1,R13              Point to my save area               00720036
         L     R13,4(R1)           Rescue pointer to high save area    00730036
*                                                                       00740036
         STORAGE RELEASE,SP=(15),ADDR=(1),LENGTH=(0)                   00750036
*                                                                       00760036
         LR    R15,R10             Reinstate return code               00770036
         RETURN (14,12),RC=(15)    Return                              00780036
*                                                                       00790036
Encr     DC    CL4'Encr'                                               00800036
Encr000  DS    0H                                                      00810036
         STM   R0,R15,WkSaveA      Store registers                     00820036
         MVC   WkPhase,Encr        Set phase name                      00830036
         XC    WkRetcde,WkRetcde   Clear return code                   00840036
*                                                                       00850036
         LM    R7,R11,0(R1)        R7 to R11 point to my parameters    00860036
         STM   R7,R11,WkParms      Store parms for debugging etc.      00870036
*                                                                       00880036
         XR    R5,R5               Clear Length                        00890036
         ICM   R5,B'0011',2(R8)    Insert length from Option Block     00900036
         BZ    Encr720             Too short, error                    00910036
*                                                                       00920036
         C     R5,=F'32767'        Test Data Length                    00930036
         BH    Encr720             Too Long, error                     00940036
*                                                                       00950036
         LR    R2,R10              Set target address                  00960036
         LR    R4,R9               Set source address                  00970036
*                                                                       00980036
         XC    WkReg0,WkReg0       Clear Reg0 field                    00990036
         CLI   1(R8),C'E'          Is this Encrypt?                    01000036
         BE    Encr020             Yes, continue                       01010036
*                                                                       01020036
         OC    WkReg0,=X'00000080' Turn on Decipher bit                01030036
```

```
         CLI   1(R8),C'D'          Is this Decrypt                    01040036
         BNE   Encr700             No, invalid                        01050036
*                                                                     01060036
Encr020  DS    0H                                                     01070036
         LA    R3,8                Assume DEA(DES) operation          01080036
         LA    R6,8                Assume Single length Key           01090036
         CLI   0(R8),X'01'         Is this Single DES                 01100036
         BE    Encr050             Yes, proceed.                      01110036
*                                                                     01120036
         LA    R6,32               Single length + Ver Patt           01130038
         CLI   0(R8),X'09'         Is this Single DES                 01140038
         BE    Encr050             Yes, proceed.   (Prot Key)         01150038
*                                                                     01160038
         LA    R6,16               Assume Double length Key           01170038
         CLI   0(R8),X'02'         Is this Triple DES, Double Len Key 01180038
         BE    Encr050             Yes, proceed.                      01190038
*                                                                     01200038
         LA    R6,40               Double length + Ver Patt           01210038
         CLI   0(R8),X'0A'         Is this Triple DES, Double Len Key 01220038
         BE    Encr050             Yes, proceed.   (Prot Key)         01230038
*                                                                     01240038
         LA    R6,24               Assume Triple length Key           01250038
         CLI   0(R8),X'03'         Is this Triple DES, Triple Len Key 01260038
         BE    Encr050             Yes, proceed.                      01270038
*                                                                     01280038
         LA    R6,48               Triple length + Ver Patt           01290038
         CLI   0(R8),X'0B'         Is this Triple DES, Triple Len Key 01300038
         BE    Encr050             Yes, proceed.   (Prot Key)         01310038
*                                                                     01320038
         LA    R3,16               Assume AES operation               01330038
         LA    R6,16               Assume AES-128                     01340038
         CLI   0(R8),X'12'         Is this AES-128 ?                  01350038
         BE    Encr050             Yes, proceed.                      01360038
*                                                                     01370038
         LA    R6,48               Assume AES-128 + Ver Pattern       01380038
         CLI   0(R8),X'1A'         Is this AES-128 ? (Prot Key)       01390038
         BE    Encr050             Yes, proceed.                      01400038
*                                                                     01410038
         LA    R6,24               Assume AES-192                     01420038
         CLI   0(R8),X'13'         Is this AES-192 ?                  01430038
         BE    Encr050             Yes, proceed.                      01440038
*                                                                     01450038
         LA    R6,56               Assume AES-192 + Ver Pattern       01460038
         CLI   0(R8),X'1B'         Is this AES-192 ? (Prot Key)       01470038
         BE    Encr050             Yes, proceed.                      01480038
*                                                                     01490038
         LA    R6,32               Assume AES-256                     01500038
         CLI   0(R8),X'14'         Is this AES-256 ?                  01510038
         BE    Encr050             Yes, proceed.                      01520038
*                                                                     01530038
         LA    R6,64               Assume AES-256 + Ver Pattern       01540038
         CLI   0(R8),X'1C'         Is this AES-256 ? (Prot Key)       01550038
         BNE   Encr710             No, bad retcode                    01560038
*                                                                     01570038
Encr050  DS    0H                                                     01580038
```

```
         OC    WkReg0+3(1),0(R8)   Insert 7-bit Function code          01590038
         L     R0,WkReg0           Load Function Code                  01600038
*                                                                      01610038
         XC    WkKMCPrm,WkKMCPrm   Clear parameter block               01620038
         LA    R1,WkKMCPrm(R3)     Point to Key Area                   01630038
*                                                                      01640038
         BCTR  R6,R0               Reduce length by 1                  01650038
         MVC   0(1,R1),0(R11)      Move Key                            01660038
         EX    R6,*-6              Move Key                            01670038
*                                                                      01680038
Encr100  DS    0H                                                      01690038
         LA    R1,WkKMCPrm         Set Parm Block Address              01700038
         KMC   R2,R4               Perform operation                   01710038
*                                                                      01720038
         BM    Encr730             Verification Pattern Mismatch       01730038
         BNZ   Encr100             Loop back if partial completion     01740038
         BZ    Encr900                                                 01750038
*                                                                      01760038
Encr700  DS    0H                                                      01770038
         MVC   WkRetcde+1(1),1(R8) Save failing byte                   01780038
         MVI   WkRetcde+2,1        Set reason                          01790038
         B     Encr800                                                 01800038
*                                                                      01810038
Encr710  DS    0H                                                      01820038
         MVC   WkRetcde+1(1),0(R8) Save failing option code            01830038
         MVI   WkRetcde+2,2        Set reason                          01840038
         B     Encr800                                                 01850038
*                                                                      01860038
Encr720  DS    0H                                                      01870038
         STCM  R5,B'0011',WkRetcde Save failing length                 01880038
         MVI   WkRetcde+2,3        Set reason                          01890038
         B     Encr800                                                 01900038
*                                                                      01910038
Encr730  DS    0H                                                      01920038
         MVI   WkRetcde+2,4        Set reason (Mismatch Ver Patt)      01930038
         B     Encr800                                                 01940038
*                                                                      01950038
Encr800  DS    0H                                                      01960038
         MVI   WkRetcde+3,12       Set retcode 12                      01970038
*                                                                      01980038
Encr900  DS    0H                                                      01990038
         MVC   0(4,R7),WkRetcde    Store RC into callers parmlist      02000038
         LM    R0,R15,WkSaveA      Reload registers                    02010038
         BR    R14                 Return to caller                    02020038
*                                                                      02030038
         DROP  R12                 CPACF010                            02040038
         DROP  R13                 #Work                               02050038
         LTORG                                                         02060038
*                                                                      02070038
         TITLE '***** CPACF010 : EQUATES *****'                        02080038
*                                                                      02090038
R0       EQU   0                                                       02100038
R1       EQU   1                                                       02110038
R2       EQU   2                                                       02120038
R3       EQU   3                                                       02130038
```

```
R4       EQU   4                                                    02140038
R5       EQU   5                                                    02150038
R6       EQU   6                                                    02160038
R7       EQU   7                                                    02170038
R8       EQU   8                                                    02180038
R9       EQU   9                                                    02190038
R10      EQU   10                                                   02200038
R11      EQU   11                                                   02210038
R12      EQU   12                                                   02220038
R13      EQU   13                                                   02230038
R14      EQU   14                                                   02240038
R15      EQU   15                                                   02250038
         TITLE '***** CPACF010 : WORK AREA *****'                   02260038
#Work    DSECT                                                      02270038
WkStart  EQU   *                                                    02280038
WkOsSave DS    18F                OS save area                     02290038
WkLen    DS    F                  Length                           02300038
WkSpl    DS    F                  Subpool                          02310038
*                                                                   02320038
WkPhase  DS    CL4                Phase name                       02330038
WkSaveA  DS    16F                Subroutine save area 1           02340038
*                                                                   02350038
WkParms  DS    5F                 Parm addresses                   02360038
*                                                                   02370038
WkRetcde DS    F                  Return Code                      02380038
*                                                                   02390038
WkReg0   DS    F                  Work area                        02400038
*                                                                   02410038
WkKMCPrm DS    XL80               Max length of KMC parameter block 02420039
*                                                                   02430038
WkEnd    EQU   *                                                    02440038
*                                                                   02450038
         END                                                        02460038
```

REXCP050 REXX program

This program is used to perform protected key encryption and then use clear key encryption using the same key to verify results. This program makes use of the two data sets used by the CPACF001 JCL:

- LENNIE.PROTECT.KEY
- LENNIE.LABEL.KEY

```
/* Rexx */                                                              00010000
/*--------------------------------------------------------------------*/ 00020000
/*                                                                    */ 00030000
/*   Invoke CPACF010 to perform Protected key encryption              */ 00040000
/*                                                                    */ 00050000
/*--------------------------------------------------------------------*/ 00060000
                                                                        00070000
/* trace i                                                  */          00080000
say ' ';                                                                00090000
say ' ';                                                                00100000
say ' ';                                                                00110000
say ' ';                                                                00120000
say '-----------------------------------------------------';            00130000
say ' ';                                                                00140000
say ' Running CPACF010 to perform AES Protected Key Encryption'         00150000
say ' ';                                                                00160000
                                                                        00170000
parm1 = '00000000'X                                                     00180000
parm2 = '1C'X||'E'||'0030'X                                             00190000
parm3 = 'Bonjour matelot. Comment ca va. Tres bien merci.'              00200000
parm4 = COPIES('00'X,48)                                                00210000
parm5 = COPIES('00'X,64)                                                00220000
                                                                        00230000
"ALLOC F(PROTECT) DA('LENNIE.PROTECT.KEY') SHR REUSE"                   00240002
"EXECIO 1 DISKR PROTECT (FINIS STEM pro."                               00250002
"FREE F(PROTECT)"                                                       00260002
                                                                        00270001
Parm5 = SUBSTR(pro.1,1,64)                                              00280000
Say '        AES Key = '||c2x(substr(Parm5,1,32))                       00290000
Say '    Ver Pattern = '||c2x(substr(Parm5,32,32))                      00300000
Say 'Data to encrypt = '||Parm3                                         00310000
Say 'Data to encrypt = '||c2x(Parm3)                                    00320000
Say ' Length of data = '||length(Parm3)                                 00330000
Say ' '                                                                 00340000
                                                                        00350000
address linkpgm 'CPACF010',                                             00360000
                'parm1',                                                00370000
                'parm2',                                                00380000
                'parm3',                                                00390000
                'parm4',                                                00400000
                'parm5',                                                00410000
                                                                        00420000
/* check the return and reason codes */                                 00430000
                                                                        00440000
if (parm1 <> '00000000'X) then do                                       00450000
  say 'CPACF010 FAILed : RC =' c2x(parm1);                              00460000
```

```
  say ' ' ;                                                          00470000
  say c2X(parm1)                                                     00480000
  say c2X(parm2)                                                     00490000
  say c2X(parm5)                                                     00500000
  end ;                                                              00510000
else do ;                                                            00520000
  say 'CPACFO10 OK '                                                 00530000
  say ' ' ;                                                          00540000
  say 'Clear_Text  (part1)...'||c2x(SUBSTR(parm3,01,16))             00550000
  say 'Cipher_Text (part1)...'||c2x(SUBSTR(parm4,01,16))             00560000
  say ' ' ;                                                          00570000
  say 'Clear_Text  (part2)...'||c2x(SUBSTR(parm3,17,16))             00580000
  say 'Cipher_Text (part2)...'||c2x(SUBSTR(parm4,17,16))             00590000
  say ' ' ;                                                          00600000
  say 'Clear_Text  (part3)...'||c2x(SUBSTR(parm3,33,16))             00610000
  say 'Cipher_Text (part3)...'||c2x(SUBSTR(parm4,33,16))             00620000
  say ' ' ;                                                          00630000
  say 'CPACFO10 completed'                                           00640000
  end ;                                                              00650000
                                                                     00660000
/*--------------------------------------------------------------------*/ 00670000
/*                                                                    */ 00680000
/*   Invoke CSNBSYD                                                   */ 00690000
/*                                                                    */ 00700000
/*--------------------------------------------------------------------*/ 00710000
                                                                     00720000
/* ************************************************** */           00730000
/* initialize parameters for common use                 */           00740000
/*                                                      */           00750000
                                                                     00760000
/* trace i                                              */           00770000
say ' ';                                                             00780000
say ' ';                                                             00790000
say '-----------------------------------------';                     00800000
say ' ';                                                             00810000
say 'Running CSNBSYD to perform AES Decryption'                      00820000
say ' ';                                                             00830000
                                                                     00840000
"ALLOC F(LABEL) DA('LENNIE.LABEL.KEY') SHR REUSE"                    00850002
"EXECIO 1 DISKR LABEL (FINIS STEM LAB."                              00860002
"FREE F(LABEL)"                                                      00870002
                                                                     00880002
label = SUBSTR(LAB.1,1,64)                                           00890000
Say 'Key = '||label                                                  00900000
Say ' '                                                              00910000
                                                                     00920000
ex_rc = '00000000'X                                                  00930000
ex_rs = '00000000'X                                                  00940000
exit_data_length = '00000000'X                                       00950000
exit_data        = '        '                                        00960000
rule_array_count = '00000003'x                                       00970000
rule_array       = 'AES     KEYIDENTINITIAL '                        00980000
key_length       = '00000040'x  /* 64 for KEYIDENT */               00990000
key_identifier   = label                                             01000000
key_parms_len    = '00000004'x                                       01010000
```

```
key_parms        = '    '                                                  01020000
block_size       = '00000010'x                                             01030000
init_vector_len  = '00000010'x                                             01040000
init_vector      = '00000000000000000000000000000000'X                     01050000
chain_data_len   = '00000020'x                                             01060000
chain_data       = COPIES('00'X,32)                                        01070000
clear_text_len   = '00000030'x                                             01080000
clear_text       = COPIES('00'X,48)                                        01090000
cipher_text_len  = '00000030'x                                             01100000
cipher_text      = parm4                                                   01110000
opt_data_len     = '00000004'x                                             01120000
opt_data         = '00000000'X                                             01130000
                                                                           01140000
address linkpgm 'CSNBSYD',                                                 01150000
                'ex_rc',                                                   01160000
                'ex_rs',                                                   01170000
                'exit_data_length',                                        01180000
                'exit_data',                                               01190000
                'rule_array_count',                                        01200000
                'rule_array',                                              01210000
                'key_length',                                              01220000
                'key_identifier',                                          01230000
                'key_parms_len',                                           01240000
                'key_parms',                                               01250000
                'block_size',                                              01260000
                'init_vector_len',                                         01270000
                'init_vector',                                             01280000
                'chain_data_len',                                          01290000
                'chain_data',                                              01300000
                'cipher_text_len',                                         01310000
                'cipher_text',                                             01320000
                'clear_text_len',                                          01330000
                'clear_text',                                              01340000
                'opt_data_len',                                            01350000
                'opt_data'                                                 01360000
/* check the return code */                                                01370000
                                                                           01380000
if (ex_rc <> '00000000'X) | (ex_rs <> '00000000'X) then do                 01390000
  say 'CSNBSYD FAILed : RC =' c2x(ex_rc);                                  01400000
  say '                 RS =' c2x(ex_rs);                                  01410000
  say ' ' ;                                                                01420000
  end ;                                                                    01430000
else do ;                                                                  01440000
  say 'CSNBSYD OK '                                                        01450000
  say ' '                                                                  01460000
  say 'Cipher_Text (part1)...'||c2x(SUBSTR(cipher_text,01,16))             01470000
  say 'Clear_Text  (part1)...'||c2x(SUBSTR(clear_text,01,16))              01480000
  say ' ' ;                                                                01490000
  say 'Cipher_Text (part2)...'||c2x(SUBSTR(cipher_text,17,16))             01500000
  say 'Clear_Text  (part2)...'||c2x(SUBSTR(clear_text,17,16))              01510000
  say ' ' ;                                                                01520000
  say 'Cipher_Text (part3)...'||c2x(SUBSTR(cipher_text,33,16))             01530000
  say 'Clear_Text  (part3)...'||c2x(SUBSTR(clear_text,33,16))              01540000
  say ' ' ;                                                                01550000
  say 'Clear_Text = 'clear_text                                            01560000
```

```
  Say 'Clear_Text length = '||length(clear_text)                        01570000
  say ' ' ;                                                             01580000
  say 'CSNBSYD completed'                                               01590000
  end ;                                                                 01600000
                                                                        01610000
if parm3 == clear_text then do                                          01620000
  say ' '                                                               01630000
  say 'Data decrypted by CSNBSYD using clear key, exactly matches'      01640000
  say 'original data encrypted by CSNBSYE using protected key.'         01650000
  end                                                                   01660000
exit                                                                    01670000
```

REXIQF program

This program demonstrates how to use the CSFIQF interface to display information from the ICSF environment.

```
/* REXX REXX REXX REXX REXX REXX REXX REXX REXX REXX REXX REXX REXX */
/*--------------------------------------------------------------------*/
/*                                                                    */
/*  CSFIQF - retrieve ICSF and Coprocessor status                     */
/*                                                                    */
/*--------------------------------------------------------------------*/
trace o
test = n

/* Skip to new page */
say ' ' ; say ' ' ; say ' ' ; say ' ' ;

say MVSVAR(SYSNAME) 'AT' TIME() 'ON' date()' ---- ICSF Details'
say ' '
erc                    = '00000000'x ;
ers                    = '00000000'x ;
ers_4                  = '00000004'x ;
/*--------------------------------------------------------------------*/
/* CREATE THE CALL TO CSFIQF ROUTINE                                  */
/*                                                                    */
/*--------------------------------------------------------------------*/
/*                                                                    */
   iqf_rc                 = 'FFFFFFFF'x ;
   iqf_rs                 = 'FFFFFFFF'x ;
   exit_data_length       = '00000000'x ;
   exit_data              = '' ;
   rule_array_count       = '00000002'x ;
   rule_array             = 'ANY     ICSFSTAT'
   ret_data_len           = '00000100'x ;
   ret_data               = COPIES('00'x,256) ;
   res_data_len           = '00000000'x ;
   res_data               = COPIES('00'x,256) ;

address linkpgm 'CSFIQF' ,
                'iqf_rc' ,
                'iqf_rs' ,
                'exit_data_length' ,
                'exit_data' ,
                'rule_array_count' ,
                'rule_array' ,
                'ret_data_len' ,
                'ret_data' ,
                'res_data_len' ,
                'res_data' ;

   if test = y then do
     say ''
     say 'Following is from CSFIQF with ICSFSTAT'
     end
```

```
/* Check return and reason code                                        */
   if (iqf_rc ¬= erc | iqf_rs ¬= ers) then do ;
      say 'IQF interface failed, rc =' C2X(iqf_rc) ,
          'rs =' C2X(iqf_rs) ;
      say ' ' ;
      EXIT;
      END ;
   else do ;
      if test = y then do
        say ''
        say 'IQF ok, RC=' C2X(IQF_RC) ,
            'RS =' C2X(IQF_RS)
        say ' ' ;
        say 'Len ' C2X(ret_data_len) ;
        say 'Data' ret_data ;
        end
      parse value ret_data with    1  fmid    ,
                                   9  stat1   ,
                                  17  stat2   ,
                                  25  cpacf   ,
                                  33  aes     ,
                                  41  dsa     ,
                                  49  rsasig  ,
                                  57  rsakm   ,
                                  65  rsakg   ,
                                  73  accel   ,
                                  81  res1    ,
                                  89  res2
      say 'FMID               =' fmid     ;

      status1.0 ='ICSF is STARTED'
      status1.1 ='ICSF is INITIALISED'
      status1.2 ='ICSF has a valid SYM-MK'
      status1.3 ='ICSF has a SYM-MK & PKA Callable services'
      stat1=STRIP(stat1)
      if SYMBOL('status1.stat1') = 'VAR' then stat1 = status1.stat1
                       else stat1 = 'UNKNOWN VALUE' stat1
      say 'ICSF status 1       =' stat1   ;

      status2.0 ='64-bit callers not supported'
      status2.1 ='64-bit callers supported'
      status2.2 ='64-bit callers supported (TKDS specified)'
      stat2=STRIP(stat2)
      if SYMBOL('status2.stat2') = 'VAR' then stat2 = status2.stat2
                       else stat2 = 'unknown value' stat2
      say 'ICSF status 2       =' stat2   ;

      cpacf.0 ='Not available'
      cpacf.1 ='SHA-1 only'
      cpacf.2 ='DES/TDES enabled'
      cpacf.3 ='SHA-224, SHA-256 enabled'
      cpacf.4 ='SHA-224, SHA-256, and DES/TDES enabled'
      cpacf.5 ='SHA-384, SHA-512 enabled'
      cpacf.6 ='SHA-384, SHA-512, and DES/TDES enabled'
      cpacf.7 ='Encrypted CPACF functions available'
```

```
cpacf=STRIP(cpacf)
if SYMBOL('cpacf.cpacf') = 'VAR' then cpacf = cpacf.cpacf
                  else cpacf = 'unknown value' cpacf
say 'CPACF               =' cpacf ;

aes.0 ='Not available'
aes.1 ='Software only'
aes.2 ='AES-128 available'
aes.3 ='AES-256 available'
aes=STRIP(aes)
if SYMBOL('aes.aes') = 'VAR' then aes = aes.aes
                  else aes = 'unknown value' aes
say 'AES                 =' aes ;

dsa.0 ='Not available'
dsa.1 ='1024 key available'
dsa.2 ='2048 key available'
dsa=STRIP(dsa)
if SYMBOL('dsa.dsa') = 'VAR' then dsa = dsa.dsa
                  else dsa = 'unknown value' dsa
say 'DSA                 =' dsa ;

rsasig.0 ='Not available'
rsasig.1 ='1024 Key available'
rsasig.2 ='2048 Key available'
rsasig.3 ='4096 Key available'
rsasig=STRIP(rsasig)
if SYMBOL('rsasig.rsasig') = 'VAR' then rsasig = rsasig.rsasig
                  else rsasig = 'unknown value' rsasig
say 'RSA signature       =' rsasig ;

rsakm.0 ='Not available'
rsakm.1 ='1024 Key available'
rsakm.2 ='2048 Key available'
rsakm.3 ='4096 Key available'
rsakm=STRIP(rsakm)
if SYMBOL('rsakm.rsakm') = 'VAR' then rsakm = rsakm.rsakm
                  else rsakm = 'unknown value' rsakm
say 'RSA key management  =' rsakm;

rsakg.0 ='Not available'
rsakg.1 ='2048 bit modulus available'
rsakg.2 ='4096 bit modulus available'
rsakg=STRIP(rsakg)
if SYMBOL('rsakg.rsakg') = 'VAR' then rsakg = rsakg.rsakg
                  else rsakg = 'unknown value' rsakg
say 'RSA key generate    =' rsakg;

accel.0 ='Not available'
accel.1 ='At least one available for application use'
accel=strip(accel)
if SYMBOL('accel.accel') = 'VAR' then accel = accel.accel
                  else accel = 'unknown value' accel
say 'Accelerators        =' accel
end ;
```

```
   iqf_rc                = 'FFFFFFFF'x ;
   iqf_rs                = 'FFFFFFFF'x ;
   exit_data_length      = '00000000'x ;
   exit_data             = '' ;
   rule_array_count      = '00000001'x ;
   rule_array            = 'ICSFST2 '
   ret_data_len          = '00000100'x ;
   ret_data              = COPIES('00'x,256) ;
   res_data_len          = '00000000'x ;
   res_data              = COPIES('00'x,256) ;

   address linkpgm 'CSFIQF' ,
                   'iqf_rc' ,
                   'iqf_rs' ,
                   'exit_data_length' ,
                   'exit_data' ,
                   'rule_array_count' ,
                   'rule_array' ,
                   'ret_data_len' ,
                   'ret_data' ,
                   'res_data_len' ,
                   'res_data' ;

   if test = y then do
     say ''
     say 'Following is from CSFIQF with ICSFST2 '
     end

/* Check return and reason code                                        */
   if (iqf_rc ¬= erc | iqf_rs ¬= ers) then do ;
      say ''
      say 'IQF interface failed, rc =' C2X(iqf_rc) ,
          'rs =' C2X(iqf_rs) ;
      say ''
      EXIT
      end
   else do ;
      if test = y then do
        say ''
        say 'IQF ok, RC=' C2X(IQF_RC) ,
            'RS =' C2X(IQF_RS)
        say ''
        say 'Len ' C2X(ret_data_len)
        say 'Data' ret_data
        end
      parse value ret_data with    1  version ,
                                   9  fmid    ,
                                  17  stat1  ,
                                  25  stat2  ,
                                  33  stat3  ,
                                  41  stat4  ,
                                  49  stat5  ,
                                  57  res1
```

```
status1.0 ='PKA callable services disabled'
status1.1 ='PKA callable services enabled'
stat1=STRIP(stat1)
if SYMBOL('status1.stat1') = 'VAR' then stat1 = status1.stat1
                  else stat1 = 'UNKNOWN VALUE' stat1
say 'PKA status          =' stat1   ;

status2.0 ='PCKS #11 not available'
status2.1 ='PKCS #11 available'
stat2=STRIP(stat2)
if SYMBOL('status2.stat2') = 'VAR' then stat2 = status2.stat2
                  else stat2 = 'unknown value' stat2
say 'PKCS #11 status     =' stat2   ;

status3.0 ='ICSF started'
status3.1 ='ICSF initialised'
status3.2 ='AES master key valid'
stat3=STRIP(stat3)
if SYMBOL('status3.stat3') = 'VAR' then stat3 = status3.stat3
                  else stat3 = 'unknown value' stat3
say 'AES-MK status       =' stat3   ;

status4.0 ='Secure AES not available'
status4.1 ='Secure AES available'
stat4=STRIP(stat4)
if SYMBOL('status4.stat4') = 'VAR' then stat4 = status4.stat4
                  else stat4 = 'unknown value' stat4
say 'Secure AES status   =' stat4   ;

do i = 1 to 7
  substat5.i = SUBSTR(stat5,i,1)
  end

/* Key store policy for CKDS        */
select
 when substat5.1 = 0 then do
   ss5101 = 'Controls not enabled'
   say 'CKDS Key Token Auth =' ss5101
   end
 when substat5.1 = 1 then do
   ss5111 = 'Controls enabled in FAIL mode'
   ss5112 = 'Key store policy ACTIVE'
   say 'CKDS Key Token Auth =' ss5111
   say '                     ' ss5112
   end
 when substat5.1 = 2 then do
   ss5121 = 'Controls enabled in WARN mode'
   ss5122 = 'Key store policy ACTIVE'
   say 'CKDS Key Token Auth =' ss5121
   say '                     ' ss5122
   end
 when substat5.1 = 3 then do
   ss5131 = 'Controls enabled in FAIL mode'
   ss5132 = 'Key store policy ACTIVE'
```

```
   ss5133 = 'Default key label checking enabled'
   say 'CKDS Key Token Auth =' ss5131
   say '                      ' ss5132
   say '                      ' ss5133
   end
 when substat5.1 = 4 then do
   ss5141 = 'Controls enabled in WARN mode'
   ss5142 = 'Key store policy ACTIVE'
   ss5143 = 'Default key label checking enabled'
   say 'CKDS Key Token Auth =' ss5141
   say '                      ' ss5142
   say '                      ' ss5143
   end
 otherwise
 end

/* Duplicate key checking for CKDS  */
select
 when substat5.2 = 0 then do
   ss52O1 = 'Controls not enabled'
   say 'CKDS Dup Key Checks =' ss52O1
   end
 when substat5.2 = 1 then do
   ss5211 = 'Controls enabled'
   ss5212 = 'Key store policy ACTIVE'
   say 'CKDS Dup Key Checks =' ss5211
   say '                      ' ss5212
   end
 otherwise
 end

/* Key store policy for PKDS        */
select
 when substat5.3 = 0 then do
   ss53O1 = 'Controls not enabled'
   say 'PKDS Key Token Auth =' ss53O1
   end
 when substat5.3 = 1 then do
   ss5311 = 'Controls enabled in FAIL mode'
   ss5312 = 'Key store policy ACTIVE'
   say 'PKDS Key Token Auth =' ss5311
   say '                      ' ss5312
   end
 when substat5.3 = 2 then do
   ss5321 = 'Controls enabled in WARN mode'
   ss5322 = 'Key store policy ACTIVE'
   say 'PKDS Key Token Auth =' ss5321
   say '                      ' ss5322
   end
 when substat5.3 = 3 then do
   ss5331 = 'Controls enabled in FAIL mode'
   ss5332 = 'Key store policy ACTIVE'
   ss5333 = 'Default key label checking enabled'
   say 'PKDS Key Token Auth =' ss5331
   say '                      ' ss5332
```

```
   say '                     ' ss5333
   end
 when substat5.3 = 4 then do
   ss5341 = 'Controls enabled in WARN mode'
   ss5342 = 'Key store policy ACTIVE'
   ss5343 = 'Default key label checking enabled'
   say 'PKDS Key Token Auth =' ss5341
   say '                     ' ss5342
   say '                     ' ss5343
   end
 otherwise
 end

/* Duplicate key checking for PKDS  */
select
 when substat5.4 = 0 then do
   ss5401 = 'Controls not enabled'
   say 'PKDS Dup Key Checks =' ss5401
   end
 when substat5.4 = 1 then do
   ss5411 = 'Controls enabled'
   ss5412 = 'Key store policy ACTIVE'
   say 'PKDS Dup Key Checks =' ss5411
   say '                     ' ss5412
   end
 otherwise
 end

/* Granular Key label access controls */
select
 when substat5.5 = 0 then do
   ss5501 = 'Controls not enabled'
   say 'Granular Key Checks =' ss5501
   end
 when substat5.5 = 1 then do
   ss5511 = 'Controls enabled in FAIL mode'
   say 'Granular Key Checks =' ss5511
   end
 when substat5.5 = 2 then do
   ss5521 = 'Controls enabled in FAIL mode'
   say 'Granular Key Checks =' ss5521
   end
 otherwise
 end

/* Symmetric label export controls    */
select
 when substat5.6 = 0 then do
   ss5601 = 'Controls not enabled'
   say 'Symmetric Export    =' ss5601
   end
 when substat5.6 = 1 then do
   ss5611 = 'Controls enabled for DES keys'
   say 'Symmetric Export    =' ss5611
   end
```

```
      when substat5.6 = 2 then do
        ss5621 = 'Controls enabled for AES keys'
        say 'Symmetric Export    =' ss5621
        end
      when substat5.6 = 3 then do
        ss5631 = 'Controls enabled for DES and AES keys'
        say 'Symmetric Export    =' ss5631
        end
      otherwise
      end

     /* Symmetric label export controls    */
     select
      when substat5.7 = 0 then do
        ss5701 = 'Controls not enabled'
        say 'PKA Mgmt Extensions =' ss5701
        end
      when substat5.7 = 1 then do
        ss5711 = 'Controls enabled in FAIL mode'
        ss5712 = 'Trusted repository is a SAF keyring'
        say 'PKA Mgmt Extensions =' ss5711
        say '                     ' ss5712
        end
      when substat5.7 = 2 then do
        ss5721 = 'Controls enabled in FAIL mode'
        ss5722 = 'Trusted repository is a PKCS #11 token'
        say 'PKA Mgmt Extensions =' ss5721
        say '                     ' ss5722
        end
      when substat5.7 = 3 then do
        ss5731 = 'Controls enabled in WARN mode'
        ss5732 = 'Trusted repository is a SAF keyring'
        say 'PKA Mgmt Extensions =' ss5731
        say '                     ' ss5732
        end
      when substat5.7 = 4 then do
        ss5741 = 'Controls enabled in WARN mode'
        ss5742 = 'Trusted repository is a PKCS #11 token'
        say 'PKA Mgmt Extensions =' ss5741
        say '                     ' ss5742
        end
      otherwise
      end

     end
EXIT
```

REXIQA program

This program demonstrates how to use the CSFIQA interface to display information about the supported algorithms in the ICSF environment.

```
/* REXX REXX REXX REXX REXX REXX REXX REXX REXX REXX REXX REXX REXX */
/*--------------------------------------------------------------------*/
/*                                                                    */
/*  CSFIQA - retrieve ICSF algorithms                                 */
/*                                                                    */
/*--------------------------------------------------------------------*/
trace o
test = n

/* Skip to new page */
say ' ' ; say ' ' ; say ' ' ; say ' ' ;

say MVSVAR(SYSNAME) 'AT' TIME() 'ON' date()' ---- ICSF Algorithms'
say ' '
erc                    = '00000000'x ;
ers                    = '00000000'x ;
ers_4                  = '00000004'x ;
/*--------------------------------------------------------------------*/
/* CREATE THE CALL TO CSFIQA ROUTINE                                  */
/*                                                                    */
/*--------------------------------------------------------------------*/
/*                                                                    */
   iqa_rc                 = 'FFFFFFFF'x ;
   iqa_rs                 = 'FFFFFFFF'x ;
   exit_data_length       = '00000000'x ;
   exit_data              = '' ;
   rule_array_count       = '00000000'x ;
   rule_array             = '        '
   ret_data_len           = '00000800'x ;
   ret_data               = COPIES('00'x,2048) ;
   res_data_len           = '00000000'x ;
   res_data               = COPIES('00'x,256) ;

address linkpgm 'CSFIQA' ,
                'iqa_rc' ,
                'iqa_rs' ,
                'exit_data_length' ,
                'exit_data' ,
                'rule_array_count' ,
                'rule_array' ,
                'ret_data_len' ,
                'ret_data' ,
                'res_data_len' ,
                'res_data' ;

   if test = y then do
     say ''
     say 'Following is from CSFIQA '
     end
```

```
/* Check return and reason code                                        */
   if (iqa_rc ¬= erc | iqa_rs ¬= ers) then do ;
      say 'IQA interface failed, rc =' C2X(iqa_rc) ,
          'rs =' C2X(iqa_rs) ;
      say ' ' ;
      EXIT;
      END ;
   else do ;
      if test = y then do
        say ''
        say 'IQA ok, RC=' C2X(IQA_RC) ,
            'RS =' C2X(IQA_RS)
        say ' ' ;
        say 'Len ' C2X(ret_data_len) ;
        say 'Data' ret_data ;
        end

      len = X2D(C2X(ret_data_len))
      num = len / 32
      say 'Algorithm Max      Security Implementation'
      say '---------- -------- -------- --------------'
      do i = 0 to num
        start = (i * 32) + 1
        alg = SUBSTR(ret_data,start,8)
        max = SUBSTR(ret_data,start+8,8)
        sec = SUBSTR(ret_data,start+16,8)
        imp = SUBSTR(ret_data,start+24,8)
        say alg||'  '||max||' '||sec||' '||imp
        end

  EXIT
```

Related publications

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

IBM Redbooks

For information about ordering these publications, see “How to get Redbooks” on page 301. Note that some of the documents referenced here may be available in softcopy only.

- *IBM System z10 Enterprise Class Technical Guide,* SG24-7516-01
- *IBM System z10 Business Class Technical Overview*, SG24-7632

Other publications

These publications are also relevant as further information sources:

- *z/OS ICSF Overview,* SA22-7519-14
- *z/OS ICSF Administrator’s Guide,* SA22-7521-15
- *z/OS ICSF TKE Workstation User’s Guide,* SA23-2211-06

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 publications, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Numerics

4755 Cryptographic Adapter 180
4765 Cryptographic Coprocessor 29
  4765 hardware 31
  coprocessor implementation 30
  IBM CCA functions 35
  IBM platforms 36
  role-based access control mechanism 35
  secure packaging technology 32
  software 34
    Segment 0 34
    Segment 1 34
    Segment 2 35
    Segment 3 35
  Specific usage characteristics of the 4765 coprocessor in System z 37
  System z Crypto Express3 37

A

Acceleration 21
access control points
  TKE V3.1 181
Access control to cryptographic resources and function 20
APDEDICATED 24
APVIRTUAL 24

C

C API
  PKCS#11 130
  token management callable services 130
CA certificate 129
callable services
  access authorization 131
certificate information 159
certificate object 123
certificates
  RACF panels 159
change Cryptographic controls 78
CKDS 20, 108
Common Cryptographic Architecture (CCA) 124
  PKCS#11, as alternative to 20, 121
Control domain index 183
CP Assist for Cryptographic Functions (CPACF
  Message Security Assist instructions 5
CP Assist for Cryptographic Functions (CPACF) 51–52
  feature code 3863 52
  history 55
  invoking the CPACF functions via ICSF 55
  Message-Security Assist instructions 53
  overview 5
  protected keys 61
    Crypto Express 3 device in conjunction with ICSF 62
    details about CPACF protected keys 70
    Using ICSF clear keys as protected keys 67
  supported algorithms and key lengths 6
  table of functions 55
  verify that the processor has the feature installed 52
CPACF 5, 21
CPRB (Cooperative Processing Request/Response Block) 45
CRYPTO 24
Crypto Express2
  support 26
Crypto Express3 37
  Access control for the provided functions 11
  access control points 38
  Asymmetric algorithms support 9
    Modular exponentiation 10
    RSA Digital signature function 10
    Support of RSA “retained keys” 10
    Support of the RSA key pairs management 10
    Symmetric key encryption function 10
  CCA services 39
  coprocessor mode 39
  coprocessors 8
  Data integrity using symmetric and one-way algorithms 9
    Message Authentication Code (MAC) 9
    Modification Detection Code (MDC) 9
    one-way hash algorithms 9
    SHA-1 and SHA-2 (SHA-256, SHA-384 and SHA-512) 9
  Data protection using symmetric algorithms 9
    Data encryption and decryption 9
    DES and AES key generation, importation and distribution 9
  Export ontrol 37
  Feature Code 0864 8, 37
  Feature Code 0871 8, 37
  Financial services support 10
    Credit card Card Verification Value or Card Security Code processing 10
    EMV (Europay - Master Card-Visa) protocol 10
    PIN (Personal Identification Number) processing services 10
    SET (Secure Electronic Transfer) protocol 10
  IBM 4765 coprocessor card 8
  IBM CCA (Common Cryptographic Architecture) API 8
  IBM UDX offering 43
  Implementation of customized functions in the coprocessor 11
  Logical partitioning support 38
  Master Key
    concept 7
    protection 8

Master keys 9, 38
AES Master Key (AES-MK) 9
PKA Master Key (ASYM-MK) 9
DES Master Key (DES-MK) 9
maximum number of coprocessors available on a single System z10 8
Modes of operations of the CEX3C coprocessor 11
The accelerator mode 11
The coprocessor mode 11
Other functions supported 11
overview 7
performance 46
Physical packaging and plugging location 37
secure key
concept 7
Synchronous and asynchronous operations 11
TKE workstation 41
Crypto Express3 feature in System z10 12
maximum number of CEX3C and CEX3A in a System z10 13
sharing of coprocessor by logical partition 13
Cryptographic
function support 2
processors 4
Cryptographic Coprocessor Facility 2
Cryptographic features comparison 16
Cryptographic Key Data Set 20
cryptographic services for Linux on System z 229
Cryptoki *See* PKCS#11
CryptoVSE API 22
CRYPTOZ class 125
resource definitions 134
resource name 129
CSFSERV class 131
CSNBRNG 55
CSNBSYD 55
CSNBSYE 55

D

DASD (direct access storage device) 125
DEFAULT
role 177
Default token checking 89
default value 128
DES 2
digital certificate *See* certificates
direct access storage device (DASD) 125
DOMAIN 24
domains 12
Duplicate token prevention 90

E

Elliptic Curve Cryptography (ECC)
algorithms 248
CSFPGKP (Generate Key Pair) 252
CSFPPKS (Private Key SIgn) 257
CSFPPKV (Public Key Verify) 258
CSFPTRC (Token Record Create) 251
CSFPTRD (Token Record Delete) 259
domain parameters 248
elliptic curve domain parameters 249
key generation process 248
mathematical theory 247
NIST recommendations for minimum key sizes 250
sample program 251

F

FC0855 14
FC0859 13
FC0863 13
FC0887 14
FC0888 14
Feature Code 3863 52
FIPS 140-2 2
Functional differences between the CPACF and the CEX3C coprocessor 16

H

hash 54
HCR7750 21

I

IBM 4753 11
IBM 4754 Security Interface Unit 180
IBM 4758-002 Cryptographic Adapter 180–181
IBM Personal Smart Card
TKE V2 180
IBMJSSE
tokens and 125
IBMPKCS11Impl 23
ICSF 20
AES Master Key 106
CKDS 108
CKDS and key labels 88
default token checking 93
Elliptic Curve Cryptography (ECC) 246
key label access levels 96
key store policies 87
key stores 108
key token
default token checking 89
key tokens
duplicate token prevention 90
modes of operation 85
with CPACF only 86
with Crypto Express device 85
without either CPACF or a Crypto Express device 87
PAN lengths 111
PKA key management extentions control 99
PKDS 109
preventing duplicate tokens 94
RACF resource profiles 103
RACF segment 102
RSA 107
Secure AES keys 105

started procedure 85
symmetric key export controls 98
Token Checking 92
versions since HCR7730 84
ICSF callable services
PKCS#11, provided by 130
ICSF Overview 19
ICSF panels
token management 126
ICSF releases 21
ICSF Token Management 134

J

JAVA 23

K

key store policies 87
key token 89
key types 59
clear key 59
Protected keys 60
secure key 59
Why do we need protected keys 60
wrapping keys 60
keystores
RACF and System SSL differences 124
KLMD 6, 57
KM 5, 55
KMAC 58
KMC 5, 56
KMID 6

L

libica 23
Linux for System z 229
acceleration of clear keys operations 233
CCA application management of secure keys 238
CCA library and API 234
application independent key-storage system 238
cryptographic libraries infrastructure 235
services provided by the Crypto Express3 coprocessor 236
Access Control Points 238
Asymmetric algorithms support 236
Data integrity using symmetric and one-way algorithms 236
Data protection using symmetric algorithms 236
Financial services support 237
Implementation of customized functions in the coprocessor 238
Other functions supported 237
CCA Support Program for Linux for System z 234
cryptographic services 232
clear keys only operations 232
secure key operations 232
Kernel's API 234
libica library 233
Mangement of the Crypto Express3 240
command line interface panel 241
Trusted Key Entry (TKE) 240
supported cryptographic hardware facilities 231
CP Assist for Cryptographic Functions 231
Crypto Express3 231
LPAR Cryptographic Controls 77

M

MAC 54–55
Mainframe cryptographic support history 2
Common Cryptographic Architecture 2
CP Assist for Cryptographic Functions 3
Crypto Express2 3
Crypto Express3 4
PCI Cryptographic Accelerator 3
PCI Cryptographic Coprocessor 3
PCI Extended Cryptographic Coprocessor 3
Manual-control panel 180
Message-Security Assist 52
Message-Security Assist instructions 53
Generation of Hash values 54
Generation of Message Authentication Codes 54
Pseudo Random Number Generation 55
Symmetric Encryption and Decryption 53
MSA functions 53

P

PAN lengths 111
PCI Cryptographic Candidate List 183
PCI Cryptographic Online List 183
PCICC
code structure and UDX 44
PCICC option 159
Perform Cryptographic Key Management Operation 53
performance
creating TKDS 127
PIN (personal identification number)
PKCS#11 and 123
PKCS#11 20, 23, 121
objects defined by 123
overview 123
RACF panels 158
tokens 124
PKCS#11 API and Elliptic Curve Cryptography support 20
PKCS#11 cryptography API
benefits on z/OS 124
infrastructure and setup 125
TKDS and 126
token administration 133
tokens 126
tokens, access to 128
PKDS 20, 109
PKSC 180
Private Key Data Set 20
private keys 123
PRNG 55
Profile 177

public key 123
Public Key Algorithm 10, 40, 237
Public Key Cryptography Standard #11 *See* PKCS#11 cryptography API

Q

Query 56

R

RACDCERT commands 124
  functions 158
  PKCS#11 and 158
RACF profile 126
Redbooks Web site 301
  Contact us xiii
Retained key support 11
RSA 10, 40, 237
RSA keys 123

S

sample JCL 127
secret keys
  key objects and 123
Secure AES keys 105
security officer (SO) 123
SHA-1 54
SHA-256 54
Sharing the Crytpo Express3 between logical partitions 12
SMF Common Address Space Work type 30 116
SMF ICSF record type 82 115
SMF RMF Processor Activity type 70 116
SMF Workload Activity type 72 117
SO.token-name resource 125
SSL Daemon 22
SSL for VSE API 22
started procedure for ICSF 85
SYSPLEXTKDS option 127
System Authorization Facility (SAF) 131
System SSL 124
  key database 124
  tokens and 125
System z10 cyrptographic functions 2
  CP Assist for Cryptographic Functions (CPACF) 4
  Crypto Express 3 feature (CEX3C) 4
  ICSF (Integrated Cryptographic Service Facility) 4
  Trusted Key Entry (TKE) workstation 5
System z10 G3 EC or BC cryptographic feature codes 13

T

TKDS 20
TKDS *See* Token Key Data Set
TKDSN option 127
TKE
  4758 user profile 178
  Domains 180
  host 174
  Master key 178
TKE V5.0 173
TKEUSER
  role 178
Token Key Data Set (TKDS) 125–126
  PKCS#11 and 124, 126
token management callable services 130
token names 126, 134
tokens
  managing, in z/OS 124
  PKCS#11 and 126
  PKCS#11 and, token access 128
Trusted Key Entry (TKE) workstation version 6.0 173
  admin, auditor and service modes 208
  clear the New Master Key register 222
  copy and store Host configuration data 193
  display the loaded signature key information 222
  Domain Grouping 184
  Domain groupings 183
  Group Logon feature 178
  history 180
  host role and authority changes 219
  logon 178
  Manage Print Screen option 224
  migration 227
  multiple TKE workstations 176
  optional hardware 182
  overview 174
  roles and profiles 177
  Smart Card support 175
  TKE Host Transaction Program 179
  workstation hardware 182
Trusted Key Entry workstation 14

U

UDX 11, 42
  callable service 45
  code 44, 120
  Invocation 43
UDX toolkit 43
Usage domain index 183
Usage Domain zeroize 74
User Defined Extension (UDX) 42
  function identifier 45
  functional implementation 44
  ICSF callable service 45
  Installation 44
  link-edited into the segment 3 load module 44
USER.token-name resource 125

V

VTAM APPC 180

Z

z10 Cryptographic configuration 73
  LPAR Cryptographic settings 79
  LPAR Security settings 80
  modify LPAR Cryptographic Controls 78
  Usage Domain Zeroize 74

view the LPAR Cryptographic Controls 77
z90crypt 23
zcrypt 23

System z Crypto and TKE Update

System z Crypto and TKE Update

Crypto Express3 technical overview and usage

Recent ICSF changes and usage with System z Cryptograpy

Details on TKE Version 6 features

This IBM Redbooks publication provides detailed information about the implementation of hardware cryptography in the System z10 server. We begin by summarizing the history of hardware cryptography on IBM Mainframe servers, introducing the cryptographic support available on the IBM System z10, introducing the Crypto Express3 feature, briefly comparing the functions provided by the hardware and software, and providing a high-level overview of the application programming interfaces available for invoking cryptographic support.

This book then provides detailed information about the Crypto Express3 feature, discussing at length its physical design, its function and usage details, the services that it provides, and the API exposed to the programmer. This book also provides significant coverage of the CP Assist for Cryptographic Functions (CPACF). Details on the history and purpose of the CPACF are provided, along with an overview of cryptographic keys and CPACF usage details. A chapter on the configuration of the hardware cryptographic features is provided, which covers topics such as zeroizing domains and security settings. We examine the software support for the cryptographic functions available on the System z10 server. We look at the recent changes in the Integrated Cryptographic Service Facility (ICSF) introduced with level HCR7770 for the z/OS operating system. A discussion of PKCS#11 support presents an overview of the standard and provides details on configuration and exploitation of PKCS#11 services available on the z/OS operating system.

The Trusted Key Entry (TKE) Version 6.0 workstation updates are examined in detail and examples are presented on the configuration, usage, and exploitation of the new features. We discuss the cryptographic support available for Linux on System z, with a focus on the services available through the IBM Common Cryptographic Architecture (CCA) API. We also provide an overview on Elliptical Curve Cryptography (ECC), along with examples of exploiting ECC using ICSF PKCS#11 services. Sample Rexx and Assembler code is provided that demonstrate the capabilities of CPACF protected keys.

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

SG24-7848-00 ISBN 0738435546