SafeNet ProtectToolkit-J 5.9

REFERENCE GUIDE

Document Information

Product Version	5.9
Document Part Number	007-013682-007
Release Date	08 January 2020

Revision History

Revision	Date	Reason
Rev. A	08 January 2020	Initial release

Trademarks, Copyrights, and Third-Party Software

Copyright 2009-2020 Thales. All rights reserved. Thales and the Thales logo are trademarks and service marks of Thales and/or its subsidiaries and are registered in certain countries. All other trademarks and service marks, whether registered or not in specific countries, are the property of their respective owners.

Disclaimer

All information herein is either public information or is the property of and owned solely by Thales and/or its subsidiaries who shall have and keep the sole right to file patent applications or any other kind of intellectual property protection in connection with such information.

Nothing herein shall be construed as implying or granting to you any rights, by license, grant or otherwise, under any intellectual and/or industrial property rights of or concerning any of Thales's information.

This document can be used for informational, non-commercial, internal, and personal use only provided that:

> The copyright notice, the confidentiality and proprietary legend and this full warning notice appear in all copies.

> This document shall not be posted on any publicly accessible network computer or broadcast in any media, and no modification of any part of this document shall be made.

Use for any other purpose is expressly prohibited and may result in severe civil and criminal liabilities.

The information contained in this document is provided "AS IS" without any warranty of any kind. Unless otherwise expressly agreed in writing, Thales makes no warranty as to the value or accuracy of information contained herein.

The document could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Furthermore, Thales reserves the right to make any change or improvement in the specifications data, information, and the like described herein, at any time.

Thales hereby disclaims all warranties and conditions with regard to the information contained herein, including all implied warranties of merchantability, fitness for a particular purpose, title and non-infringement. In no event shall Thales be liable, whether in contract, tort or otherwise, for any indirect, special or consequential

damages or any damages whatsoever including but not limited to damages resulting from loss of use, data, profits, revenues, or customers, arising out of or in connection with the use or performance of information contained in this document.

Thales does not and shall not warrant that this product will be resistant to all possible attacks and shall not incur, and disclaims, any liability in this respect. Even if each product is compliant with current security standards in force on the date of their design, security mechanisms' resistance necessarily evolves according to the state of the art in security and notably under the emergence of new attacks. Under no circumstances, shall Thales be held liable for any third party actions and in particular in case of any successful attack against systems or equipment incorporating Thales products. Thales disclaims any liability with respect to security for direct, indirect, incidental or consequential damages that result from any use of its products. It is further stressed that independent testing and verification by the person using the product is particularly encouraged, especially in any application in which defective, incorrect or insecure functioning could result in damage to persons or property, denial of service, or loss of privacy.

All intellectual property is protected by copyright. All trademarks and product names used or referred to are the copyright of their respective owners. No part of this document may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, chemical, photocopy, recording or otherwise without the prior written permission of Thales.

CONTENTS

PREFACE: About the SafeNet ProtectToolkit-J Reference Guide

This document provides reference material for Java software developers using SafeNet ProtectToolkit-J. It contains the following chapters:

This preface also includes the following information about this document:

For information regarding the document status and revision history, see "Document Information" on page 2.

Gemalto Rebranding

In early 2015, Gemalto completed its acquisition of SafeNet, Inc. As part of the process of rationalizing the product portfolios between the two organizations, the SafeNet name has been retained. As a result, the product names for SafeNet HSMs have changed as follows:

Old product name	New product name
ProtectServer External 2 (PSE2)	SafeNet ProtectServer Network HSM

Old product name	New product name
ProtectServer Internal Express 2 (PSI-E2)	SafeNet ProtectServer PCIe HSM
ProtectServer HSM Access Provider	SafeNet ProtectServer HSM Access Provider
ProtectToolkit C (PTK-C)	SafeNet ProtectToolkit-C
ProtectToolkit J (PTK-J)	SafeNet ProtectToolkit-J
ProtectToolkit M (PTK-M)	SafeNet ProtectToolkit-M
ProtectToolkit FM SDK	SafeNet ProtectToolkit FM SDK

> **NOTE** These branding changes apply to the documentation only. The SafeNet HSM software and utilities continue to use the old names.

Audience

This document is intended for personnel responsible for maintaining your organization's security infrastructure. This includes SafeNet ProtectToolkit users and security officers, key manager administrators, and network administrators.

All products manufactured and distributed by Thales are designed to be installed, operated, and maintained by personnel who have the knowledge, training, and qualifications required to safely perform the tasks assigned to them. The information, processes, and procedures contained in this document are intended for use by trained and qualified personnel only.

It is assumed that the users of this document are proficient with security concepts.

Document Conventions

This document uses standard conventions for describing the user interface and for alerting you to important information.

Notes

Notes are used to alert you to important or helpful information. They use the following format:

> **NOTE** Take note. Contains important or helpful information.

Cautions

Cautions are used to alert you to important information that may help prevent unexpected results or data loss. They use the following format:

CAUTION! Exercise caution. Contains important information that may help prevent unexpected results or data loss.

Warnings

Warnings are used to alert you to the potential for catastrophic data loss or personal injury. They use the following format:

WARNING Be extremely careful and obey all safety and security measures. In this situation you might do something that could result in catastrophic data loss or personal injury.

Command Syntax and Typeface Conventions

Format	Convention
bold	The bold attribute is used to indicate the following: > Command-line commands and options (Type **dir /p**.) > Button names (Click **Save As**.) > Check box and radio button names (Select the **Print Duplex** check box.) > Dialog box titles (On the **Protect Document** dialog box, click **Yes**.) > Field names (**User Name:** Enter the name of the user.) > Menu names (On the **File** menu, click **Save**.) (Click **Menu > Go To > Folders**.) > User input (In the **Date** box, type **April 1**.)
italics	In type, the italic attribute is used for emphasis or to indicate a related document. (See the *Installation Guide* for more information.)
<variable>	In command descriptions, angle brackets represent variables. You must substitute a value for command line arguments that are enclosed in angle brackets.
[**optional**] [<optional>]	Represent optional **keywords** or <variables> in a command line description. Optionally enter the keyword or <variable> that is enclosed in square brackets, if it is necessary or desirable to complete the task.
{a|b|c} {<a>|<b>|<c>}	Represent required alternate **keywords** or <variables> in a command line description. You must choose one command line argument enclosed within the braces. Choices are separated by vertical (OR) bars.
[a|b|c] [<a>|<b>|<c>]	Represent optional alternate keywords or variables in a command line description. Choose one command line argument enclosed within the braces, if desired. Choices are separated by vertical (OR) bars.

Support Contacts

If you encounter a problem while installing, registering, or operating this product, please refer to the documentation before contacting support. If you cannot resolve the issue, contact your supplier or Thales Customer Support.

Thales Customer Support operates 24 hours a day, 7 days a week. Your level of access to this service is governed by the support plan arrangements made between Thales and your organization. Please consult this support plan for further information about your entitlements, including the hours when telephone support is available to you.

Customer Support Portal

The Customer Support Portal, at https://supportportal.thalesgroup.com, is where you can find solutions for most common problems. The Customer Support Portal is a comprehensive, fully searchable database of support resources, including software and firmware downloads, release notes listing known problems and workarounds, a knowledge base, FAQs, product documentation, technical notes, and more. You can also use the portal to create and manage support cases.

> **NOTE** You require an account to access the Customer Support Portal. To create a new account, go to the portal and click on the **REGISTER** link.

Telephone

The support portal also lists telephone numbers for voice contact (Contact Us).

CHAPTER 1: Product Overview

SafeNet ProtectToolkit-J is a Cryptographic Service Provider for the Java Cryptographic Architecture (JCA) / Java Cryptographic Extension (JCE) interface. SafeNet ProtectToolkit-J implements a number of cryptographic algorithms that are supported by SafeNet's hardware encryption devices. These devices support encryption, signature generation and verification, message digests, key storage and message authentication. SafeNet ProtectToolkit-J also includes a clean-room implementation of the JCA/JCE framework, allowing for immediate use with Java 6.x, 7.x, and 8.x.

This document assumes some knowledge of the Java programming language, the JCA/JCE application programming interfaces, and some understanding of the underlying adapter interface, which is based on PKCS#11 (Cryptoki). Refer to the *SafeNet ProtectToolkit-C Administration Manual* for more information on Cryptoki. For general information on the JCA/JCE, consult:

> "JCA/JCE API Overview" on page 27

> "JCA/JCE API Tutorial" on page 88

> JCA reference material found at http://docs.oracle.com/

This document does not discuss the security properties of the various algorithms in general; please consult the standard cryptography texts for this information.

SafeNet ProtectToolkit-J is known to the JCA/JCE by the provider name **SAFENET**. To request an algorithm implemented by this provider, the string "**SAFENET**" should be passed to the **getInstance()** method.

SafeNet ProtectToolkit-J is SafeNet's Java Cryptographic Architecture (JCA) / Java Cryptographic Extension (JCE) provider. It allows Cryptographic processing using the Java development language. It requires that SafeNet ProtectToolkit-C Runtime and an appropriate Access Provider are installed.

| The **SafeNet ProtectToolkit-C Runtime** package is needed to perform Cryptoki (PKCS#11) processing. The SafeNet ProtectToolkit-C Hardware Runtime needs an Access Provider. There are two Access Provider install packages in order to operate the Runtime in a local PCIe bus or network attached remote server arrangement. | The **SafeNet ProtectToolkit-C Software Development Kit (SDK)** is provided to develop applications using PKCS#11 processing. Refer to the *SafeNet ProtectToolkit-C Administration Guide* for instructions on how to install this SDK. The SafeNet ProtectToolkit-C SDK includes the SafeNet ProtectToolkit-C Runtime as well as a Software Emulation that does not require any Access Providers. |
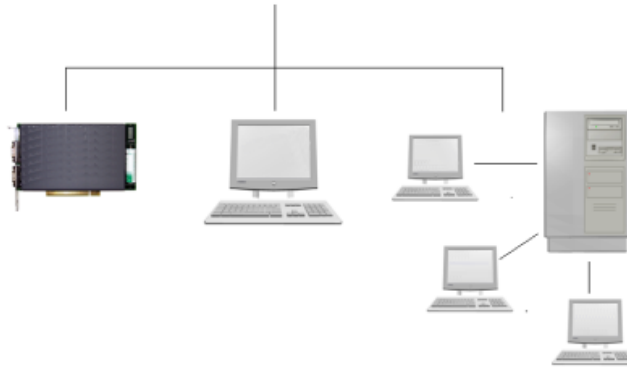
Hardware	Software	Remote Client/Server
The hardware version of SafeNet ProtectToolkit-C requires a SafeNet ProtectServer HSM. Refer to the *Installation Manuals* for instructions on how to install the adapter and the *SafeNet ProtectToolkit-C Administration Guide* for how to install the Runtime or SDK package.	The software-only version of **SafeNet ProtectToolkit-C** requires a compatible PC, and would primarily be used in a development or testing environment. Refer to the *SafeNet ProtectToolkit-C Administration Guide* for instructions on how to install the software-only version of the SDK package.	This version of SafeNet ProtectToolkit-C requires a TCP/IP network with one or more workstations and a server. SafeNet ProtectToolkit-C processing is performed by the server at the request of the client. The server must be running the Runtime package or the hardware version of the SDK package.

Working With Slots

SafeNet ProtectToolkit-J is capable of interfacing to multiple adapters. This is achieved by using different "virtual providers" which map to each adapter. The virtual providers are named **SAFENET.*n***, where *n* is the slot number as configured with the SafeNet ProtectToolkit-C runtime tools. The special provider **SAFENET** always maps to the first slot.

A provider class exists (**SAFENETProvider**) for each of the slots in the package **au.com.safenet.crypto.provider.slot<*n*>**. These providers may be statically installed. Alternatively, they may be added dynamically by calling the **SAFENETProvider.addProviders()**method.

Resource Management

Resource management is an important consideration when using the SafeNet provider. In general, creation of a provider instance (a Cipher object or Key object, for example) consumes resources within the adapter. This consumption is less than that of the main JVM and so the garbage collection is not tuned to its needs. The application programmer must therefore manage collection.

Two main techniques may be employed:

> Explicitly track resource usage, invoking garbage collector on certain thresholds. For example, after the creation of 100 "session" Key objects, which are only required for a short transaction and then discarded, it may be necessary to run the garbage collector to clean up those unused instances.

> The second technique requires some tuning of the Cryptoki configuration on the adapter. If SafeNet ProtectToolkit-J cannot create a new "session" with the adapter it invokes the garbage collection (in the hope that there are some old unused sessions awaiting cleanup). By reducing the maximum number of sessions allowed by the adapter, the adapter may be tuned to the application's requirements so that explicit resource management is not required.

System Requirements

> A SafeNet cryptographic services adapter (not required for operation in software-only mode).

> The PC on which you plan to run SafeNet ProtectToolkit-J must already have SafeNet ProtectToolkit-C Runtime installed. For system requirements, see the *SafeNet ProtectToolkit-C Administration Guide*.

> Java Runtime (for SafeNet ProtectToolkit-J Runtime) or JDK (for SafeNet ProtectToolkit-J SDK). The product has been tested using Java Runtime versions 6.x, 7.x, and 8.x. It may also operate correctly using other versions of the runtime, but this is not supported by Thales.

> **NOTE**   Java runtime or JDK must be installed before installing SafeNet ProtectToolkit-J.

The supported platforms are listed in the following table.

C=SafeNet ProtectToolkit-C, PKCS #11 v2.10/2.20

M=SafeNet ProtectToolkit-M, MS CSP 2.0 with CNG

J=SafeNet ProtectToolkit-J, Java runtime 6.x/7.x/8.x

Operating System		OS type	64-bit PTK	64-bit PTK supported hardware	32-bit PTK	32-bit PTK supported hardware
Windows	10	64-bit	C/M/J	PCIe HSM Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus
	Server 2016	64-bit	C/M/J	PCIe HSM Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus
	Server 2012 R2	64-bit	C/M/J	PCIe HSM Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus
	Server 2008 (R1 and R2)	64-bit	C/M/J	PCIe HSM Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus
	7	32-bit	-	-	C/J (KSP support)	PCIe HSM Network HSM Network HSM Plus
	7	64-bit	C/M/J	PCIe HSM Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus
Linux	RHEL 7	64-bit	C/J	PCIe HSM Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus
	RHEL 6	32-bit	-	-	C/J	PCIe HSM Network HSM Network HSM Plus
	RHEL 6	64-bit	C/J	PCIe HSM Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus
	SUSE12	64-bit	C/J	PCIe HSM Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus

Operating System		OS type	64-bit PTK	64-bit PTK supported hardware	32-bit PTK	32-bit PTK supported hardware
AIX	7.2	64-bit	C/J	Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus
	7.1	64-bit	C/J	Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus
	6.1	64-bit	C/J	Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus
Solaris	11 (SPARC, x86) 10 (SPARC, x86)	64-bit	C/J	Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus
HP-UX	11	64-bit	C/J	Network HSM Network HSM Plus	C/J	Network HSM Network HSM Plus

The Software

The latest versions of the client software and HSM firmware can be found on the Thales Technical Support Customer Portal. See "Support Contacts" on page 10 for more information. The following SafeNet ProtectToolkit-J packages can be found in the installation package:

Package	Windows	UNIX
SafeNet ProtectToolkit-J Runtime	PTKjprt.msi	PTKjprov
SafeNet ProtectToolkit-J SDK	PTKjpsdk.msi	PTKjpsdk

The **SafeNet ProtectToolkit-J Runtime** includes the necessary shared libraries required to interface to the SafeNet ProtectToolkit-C Runtime, as well as the Java class libraries that implement the JCE specification and the SafeNet ProtectToolkit-J provider. For instructions on SafeNet ProtectToolkit-C Runtime installation, please refer to the *SafeNet ProtectToolkit-C Administration Guide.*

The **SafeNet ProtectToolkit-J SDK** is provided as a software development platform.

> **NOTE**   If you will be using larger key sizes or non-FIPS algorithms, install the JCE Unlimited Strength Jurisdiction Policy Files patch. They are available for download on the Oracle website (http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html).

Installation

This chapter is intended for the administrator wqho will install SafeNet ProtectToolkit-J on host computer systems. It deals explicitly with issues and concepts involved during the SafeNet ProtectToolkit-J software installation and uninstallation.This chapter also provides instructions for installing the SafeNet ProtectToolkit-J SDK and Runtime packages. It contains the following sections:

> "Windows Installation" on the next page

> "Unix Installation Utility" on page 18

> "Unix Manual Installation" on page 22

> "Troubleshooting" on page 26

For hardware installation instructions, refer to:

> *SafeNet ProtectServer PCIe HSM Installation Guide*

> *SafeNet ProtectServer Network HSM Installation/Configuration Guide*

> *SafeNet HSM Access Provider Installation Guide*

The Provider may be statically installed into the Java Runtime Environment by adding an entry, similar to the following, into the **java.security** properties file located in **$JAVA_HOME/lib/security/java.security security.provider.2 = au.com.safenet.crypto.provider.SAFENETProvider**

Alternatively, the Provider may be installed dynamically by an application at runtime by using the **java.security.Security.addProvider()** method. For example:

Security.addProvider(new au.com.safenet.crypto.provider.SAFENETProvider());

If the Provider is to be used on a specific Slot, the format for the above references should be:

au.com.safenet.crypto.provider.slot *<n>*.SAFENETProvider

Windows Installation

Before continuing with the SafeNet ProtectToolkit-J installation, ensure that you have installed SafeNet ProtectToolkit-C Runtime. For details, please refer to the *SafeNet ProtectToolkit-C Administration Guide*.

The SafeNet ProtectToolkit-J Runtime package must be installed before the SafeNet ProtectToolkit-J Software Development Kit (SDK).

> **NOTE** To add or remove software, you must have Administrator privileges.

The installation also allows the SafeNet ProtectToolkit-J directory to be automatically added to the system's execution path. This will make the newly-installed programs and libraries available from the command prompt.

Installation File	Details
PTKjprt.msi	SafeNet ProtectToolkit-J Runtime Package
PTKjpsdk.msi	SafeNet ProtectToolkit-J SDK Package

To install the package, execute the **PTKjprt.msi** program. This will start the installation wizard. Follow the on-screen instructions to install the software. If you wish to install the SDK, repeat the process with the **PTKjpsdk.msi** program.

The installation program will create a new program group named **Safenet\ProtectToolkit J\Runtime** and add it to your **Start** menu. Program files are saved to **C:\Program Files\SafeNet\Protect Toolkit 5**.

Windows Uninstallation

To uninstall the SafeNet ProtectToolkit-J Runtime, open your **Windows Control Panel** and double-click on the **Add/Remove Program** icon. Locate the entry for the SafeNet ProtectToolkit-J installation and click on the **remove** button.

Unix Installation Utility

Before continuing with the SafeNet ProtectToolkit-J installation, ensure that you have installed SafeNet ProtectToolkit-C Runtime. For details, please refer to the *SafeNet ProtectToolkit-C Administration Guide*.

The SafeNet ProtectToolkit-J Runtime package must be installed before the SafeNet ProtectToolkit-J Software Development Kit (SDK).

> **NOTE** To add or remove software, you must first become the super-user on the host system.

Installation and uninstallation commands are different for each of the supported Unix platforms. To account for these differences, the package should be installed using the Unix Installation Utility. Manual commands specific to your operating system can be used, but this is not the recommended method. The Installation Utility is more likely to result in a problem-free installation or uninstallation. The latest versions of the client software and HSM firmware can be found on the Thales Technical Support Customer Portal. See "Support Contacts" on page 10 for more information.

The utility provides a simple menu-driven interface. In addition to installing and uninstalling the access provider on Unix systems, it can also:

> List already-installed SafeNet packages

> List directory contents, for the current platform or all platforms

> Install a package from the directory (which also installs the utility in **/usr/bin**)

> Change the default operating mode (hardware or software-only).

Whenever the utility installs a SafeNet package, it also installs itself on the host system's hard disk (in **/usr/bin/safeNet-install.sh**). This copy can be used to uninstall or configure the software.

Utility Startup

Should you encounter any problems while following this procedure, please see "Unix Installation Utility troubleshooting" on page 20. Options can be specified when executing the **safeNet-install.sh** command. These options are not normally required and are mainly useful for troubleshooting.

Syntax
safeNet-install.sh [**-h**] [**-p**] [**-s** <size>] [**-v**]

Option	Description
-h	Show help.
-p	Plain mode. In this mode the '**tput**' is not used for video enhancements.
-s<size>	Override the screen size (default = '**tput lines/cols**' or **24x80**).
-v	Print the version of this script.

If you wish to enter platform-specific commands manually, use the commands given in "Unix Manual Installation" on page 22.

To start up the utility

1. The Thales Unix Installation Utility is located in the installation DVD or image's root directory. Mount the DVD or unzip the image by following standard procedure for your platform and installation.

2. Change directory to the DVD or directory and start the utility. For example:

```
# cd /misc/cd
# ./safeNet-install.sh
```

The utility scans the system and the directory and displays the Main Menu.

```
Gemalto Unix Installation Utility (version 5.3.0):
Hostname: 66 (Linux 2.6.32-504.16.2.el6.i686)
Main menu

1 list Gemalto packages already installed
2 list packages on CD
3 install a package from this CD
4 uninstall a Gemalto package
5 Set the default cryptoki and/or hsm link

q quit the utility


Choice (1 2 3 4 q) [Redraw]:
```

> **NOTE** Enter '**b**' to go back to the previous menu and '**q**' to quit the utility. You can also quit with the system **INTR** key (normally **^C**).

Installing a package

Should you encounter any problems, please see "Unix Installation Utility troubleshooting" on the next page.

To install a package

1. Select **install a package from this CD** from the utility's Main Menu.

   A list of installable SafeNet packages is displayed.

2. Select the package required by typing the appropriate menu number followed by **Enter**.

   The utility verifies the action and executes the appropriate command for your platform.

3. On some platforms, you may be prompted for additional installation options. On Linux, for example, you can add a **–nodeps** option to suppress the checking of dependencies. These options should be selected with appropriate care.

4. You may now need to respond to any platform-specific messages (for example: to confirm you wish to proceed with the installation).

5. After installation, the utility will return **Success** or **Failure**, scan the system again, and display the current installation status. Press the **Enter** key to continue.

Uninstalling a package

Should you encounter any problems, please see "Unix Installation Utility troubleshooting" below.

To uninstall a package

1. Select **Uninstall a SafeNet package** from the utility's **Main Menu**.

   A list of installed SafeNet packages is displayed.

2. Select the required package by typing the appropriate menu number and pressing **Enter**.

   The utility verifies the action and executes the appropriate command for your platform.

3. On some platforms, you may be prompted for additional uninstallation options. On Linux, for example, you can add a **–nodeps** option to suppress the checking of dependencies. These options should be selected with appropriate care.

4. After completing uninstallation, the utility will return **Success** or **Failure**, scan the system again, and display the current installation status.

5. You may now need to respond to any platform-specific messages to confirm that you wish to proceed with the uninstallation. Press the **Enter** key to continue.

Unix Installation Utility troubleshooting

Problem	Solution
Packages to install or uninstall are not visible	If no packages are shown to install or uninstall, close the utility, check that you are logged on as **root**, and ensure your current directory is on the DVD or directory before running the utility again.
The screen is confused or does not display correctly	This utility relies on the **TERM** environment parameter when creating colors and measuring screen size, so make sure this is set correctly. The most common values are **xterm** or **vt100**. For example, to set **TERM** to **vt100**: **# TERM=vt100# export TERM** > If the screen is confused, run the utility in "plain" mode as follows: **# ./safeNet-install.sh –p** > If the size of the terminal is not correctly set by **termcap** (for example: the headings disappear off the top of the screen), override the screen size with the **-s** option: **# ./safeNet-install.sh -s 24x80** > If using an X system terminal window, do not resize the window while running the utility, as it cannot sense the change.

Problem	Solution
The backspace key does not operate correctly	On some terminals, the backspace key does not operate correctly. If, after typing a number and then backspace, the terminal returns "**2^H**" instead of an actual backspace: > Type the current **KILL** character (normally **^U**) and then enter the desired number (you will need to do this each time a backspace is required) > Exit the utility (perhaps with **^C**) and use the **stty(1)** command to correct the erase character before restarting the utility: **# stty erase ^H** where **^H** is the character created by pressing the backspace key. This will fix the problem semi-permanently, for the current session in that terminal.

Unix Manual Installation

These manual methods of installation are not recommended. For best results, see . This section provides instructions for the following operating systems:

Before continuing with the SafeNet ProtectToolkit-J installation, ensure that you have installed SafeNet ProtectToolkit-C Runtime. For details, please refer to the *SafeNet ProtectToolkit-C Administration Guide*.

The SafeNet ProtectToolkit-J Runtime package must be installed before the SafeNet ProtectToolkit-J Software Development Kit (SDK).

> **NOTE** To add or remove software, you must first become the super-user on the host system.

Linux Installation

SafeNet ProtectToolkit-J for Linux is packaged using the standard **rpm** packaging software.

Installation File	Details
PTKjprov-x.xx-y.i386	SafeNet ProtectToolkit-J Runtime Package
PTKjpsdk-x.xx-y.i386	SafeNet ProtectToolkit-J SDK Package

Use the **rpm** program to install the package(s).

To install the SafeNet ProtectToolkit-J Runtime Package

```
# cd <Linux directory on the installation CD>
# rpm -i PTKjprov-x.xx-y.i386.rpm
```
(where *x.xx-y* refers to the version of the software)

Repeat the process with the **PTKjpsdk-x.xx-y.i386** file to install the SafeNet ProtectToolkit-J SDK Package.

Once installed, the software will be ready to use under **/opt/safenet/protecttoolkit5/ptk**.

Adding directories to the required paths

To make use of the software you will need to add the **/opt/safenet/protecttoolkit5/ptk/bin** directory to your execution path and **/opt/safenet/protecttoolkit5/ptk/lib** to your library path. The following commands may be used to configure your paths for the **sh(1)** shell (please consult your Linux manual for other shells).

To add the directories to the required paths

```
# PATH=/opt/safenet/protecttoolkit5/ptk/bin:$PATH
# export PATH
# LD_LIBRARY_PATH=/opt/safenet/protecttoolkit5/ptk/lib:$LD_LIBRARY_PATH
# export LD_LIBRARY_PATH
```

Linux Uninstallation

To remove the software from your host system, use the **rpm** command with the appropriate package name as a parameter.

```
# rpm -e PTKjprov
```

Solaris Installation

SafeNet ProtectToolkit-J for Solaris is packaged using the standard Solaris packaging software.

Installation File	Details
PTKjprov	SafeNet ProtectToolkit-J Runtime Package
PTKjpsdk	SafeNet ProtectToolkit-J SDK Package

Use the **pkgadd** program to install the package(s).

To install the SafeNet ProtectToolkit-J Runtime Package

```
# cd <Solaris directory on the installation CD>
# pkgadd -d `pwd`/PTKjprov
```
Repeat the process with the **PTKjpsdk** file to install the SafeNet ProtectToolkit-J SDK Package.

Once installed, the software will be ready to use under **/opt/safenet/protecttoolkit5/ptk**.

Adding the directories to the required paths

To make use of the software, you will need to add the **/opt/safenet/protecttoolkit5/ptk/bin** directory to your execution path and **/opt/safenet/protecttoolkit5/ptk/lib** to your library path. The following commands may be used to configure your paths for the **sh(1)** shell (please consult your Solaris manual for other shells):

To add the directories to the required paths

32-bit:

```
# export PATH=/opt/safenet/protecttoolkit5/ptk/bin:$PATH
# export LD_LIBRARY_PATH=/opt/safenet/protecttoolkit5/ptk/lib:$LD_LIBRARY_PATH
```
64-bit:

```
# export PATH=/opt/safenet/protecttoolkit5/ptk/bin/sparcv9:$PATH
# export LD_LIBRARY_PATH=/opt/safenet/protecttoolkit5/ptk/lib/sparcv9
```

Solaris Uninstallation

To remove the software from your host system, use the **pkgrm** program and select the appropriate package for removal.

```
# pkgrm PTKjprov
```

IBM AIX Installation

SafeNet ProtectToolkit-J for AIX is packaged using the standard AIX packaging software.

Installation File	Details
PTKjprov	SafeNet ProtectToolkit-J Runtime Package
PTKjpsdk	SafeNet ProtectToolkit-J SDK Package

Use the **installp** program to install the package(s).

To install the SafeNet ProtectToolkit-J Runtime Package

```
# installp -acgNQqwX -d . PTKjprov.rte
```
Repeat the process with the **PTKjpsdk** file to install the SafeNet ProtectToolkit-J SDK Package.

Once installed, the software will be ready to use under **/opt/safenet/protecttoolkit5/ptk**.

> **NOTE** When installing SafeNet ProtectToolkit-J on an AIX 7.1 operating system, if you encounter the message: "Errors were detected validating the files for package PTKjpsdk.rte", it may be due to a missing user **esaadmin** in the default AIX configuration. To avoid this error, create a user named **esaadmin** before installing SafeNet ProtectToolkit-J.

Adding directories to the required paths

To use the software, you will need to add the **/opt/safenet/protecttoolkit5/ptk/bin** directory to your execution path and **/opt/safenet/protecttoolkit5/ptk/lib** to your library path. The following commands may be used to configure your paths for the **sh** shell (please consult your AIX manual for other shells).

To add directories to the required paths

```
# PATH=/opt/safenet/protecttoolkit5/ptk/bin:$PATH
# export PATH
# LIBPATH=/opt/safenet/protecttoolkit5/ptk/lib:$LIBPATH
# export LIBPATH
```

IBM AIX Uninstallation

To remove the software from your host system, simply use the **installp** program and select the appropriate package for removal.

For example:

```
# installp -u PTKjprov
```

HP-UX Installation

SafeNet ProtectToolkit-J for HP-UX is packaged using the standard HP-UX packaging software.

Installation File	Details
PTKjprov.depot	SafeNet ProtectToolkit-J Runtime Package
PTKjpsdk.depot	SafeNet ProtectToolkit-J SDK Package

Use the **swinstall** program to install the package(s).

To install the SafeNet ProtectToolkit-J Runtime Package

```
# swinstall PTKjprov.depot
```
Repeat the process with the **PTKjpsdk** file to install the SafeNet ProtectToolkit-J SDK Package.

Once installed, the software will be ready to use under **/opt/safenet/protecttoolkit5/ptk**.

Adding directories to the required paths

To use the software, you will need to add the **/opt/safenet/protecttoolkit5/ptk/bin** directory to your execution path and **/opt/safenet/protecttoolkit5/ptk/lib** to your library path. The following commands may be used to configure your paths for the **sh** shell (please consult your HP-UX manual for other shells).

To add directories to the required paths

```
PATH=$PATH:/opt/safenet/protecttoolkit5/ptk/bin
export PATH
SHLIB_PATH=$SHLIB_PATH:/opt/safenet/protecttoolkit5/ptk/lib
export SHLIB_PATH
```

HP-UX Uninstallation

To remove the software from your host system, simply use the **swremove** program and select the appropriate package for removal.

For example:

```
# swremove PTKjprov
```

CHAPTER 2: Troubleshooting

The most common problems encountered while installing SafeNet ProtectToolkit-J on SafeNet ProtectToolkit-C and the SafeNet ProtectServer HSM is that the encryption board driver is not loaded or functioning correctly.

Should you encounter any difficulties, first check that you have followed all the installation instructions in this manual, and consult your relevant Installation Guide for troubleshooting options.

Try running the **hsmstate** utility. It should report each installed HSM (PCI or Network connect) is in NORMAL Mode. If that works, then try running the **ctconf** utility. This will ensure that the Cryptoki runtime is working.

Finally, if neither resolves the issue, please contact Thales support ().

CHAPTER 3:   JCA/JCE API Overview

The purpose of this appendix is to provide an introduction to the Java APIs that provide security and cryptographic services. These are known as the Java Cryptography Architecture (JCA) and Java Cryptography Extensions (JCE).

While reading this document, it is suggested you have both the JCA/JCE API documentation at hand. The JCA documentation can be found in the Java release or online at: http://docs.oracle.com/

JCE documentation is currently available at http://docs.oracle.com/.

Finally, SafeNet ProtectToolkit-J includes a detailed reference manual detailing the specific algorithms included, and the various parameters they accept. It also includes some extensions to the base JCA/JCE API.

This document assumes the reader is familiar with the Java programming language. It contains the following chapters:

> "Encryption/Decryption" below
> "Message Digests" on page 31
> "Message Authentication Code (MAC)" on page 31
> "Authentication" on page 32
> "Key Management" on page 35
> "Error Handling and Exceptions" on page 40

Encryption/Decryption

The JCE supports encryption and decryption using symmetric algorithms (such as DES and RC4) and asymmetric algorithms (such as RSA and ElGamal). The algorithms may be stream or block ciphers, with each algorithm supporting different modes, padding or even algorithm-specific parameters.

This section details the following:

> "The Cipher Class" below
> "Cipher Input and Output Streams" on the next page
> "SealedObject" on page 29
> "Algorithm Parameters" on page 30

The Cipher Class

The basic interface used to encipher or decipher data is the **javax.crypto.Cipher** class. The class provides the necessary mechanism for encrypting and decrypting data using arbitrary algorithms from any of the installed providers.

To create a Cipher instance, use one of the **Cipher.getInstance()** methods. This method will accept a transformation string and an optional provider name. The transformation string is used to specify the encryption algorithm as well as the cipher mode and padding. The transformation is specified in the form:

> "algorithm"

> "algorithm/mode/padding"

In the first instance, we are requesting the algorithm with its default mode and padding mechanism. The second instance fully qualifies all options. For a list of support algorithms consult the provider's documentation. Some common transformations are:

> "RC4"

> "DES/CBC/PKCS5Padding"

> "RSA/ECB/PKCS1Padding"

The following code will create a cipher for performing RC4 encryption or decryption, a cipher for doing RSA in ECB mode with PKCS#1 padding provided by the ABA provider and a cipher for performing DESede encryption/decryption in CBC mode with PKCS#5 padding:

```
Cipher rc4Cipher = Cipher.getInstance("RC4");
Cipher rsaCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
Cipher desEdeCipher =
            Cipher.getInstance("DESede/CBC/PKCS5Padding");
```

Once we have a Cipher instance, we will need to initialize the Cipher for encryption or decryption. We will also need to provide a Key (see "Key Management" on page 35).

```
Key desKey, rsaKey;

desCipher.init(Cipher.ENCRYPT_MODE, desKey);
rsaCipher.init(Cipher.DECRYPT_MODE, rsaKey);
```

As you can see, the first value passed to the **Cipher.init()** method indicates whether we are initializing for encryption or decryption. The second argument provides the key to use during encryption or decryption.

There are a number of other initialization methods for providing algorithm specific parameters (such as Initialization Vectors, the number of rounds to use etc.). See "Algorithm Parameters" on page 30 for more information.

Now that our Cipher is initialized, we can start processing data. To do so we use the **Cipher.update()** and **Cipher.doFinal()** methods. The **Cipher.update()** methods may be used to incrementally process data. Once all the data is processed, one of the **Cipher.doFinal()** methods must be called.

In the simplest usage, a single **Cipher.doFinal()** call may be passed all the data:

```
byte[] plainText = "hello world".getBytes();
byte[] cipherText = desCipher.doFinal(plainText);
```

Once the **Cipher.doFinal()** method has been called, the Cipher instance will be reset to the state it was in after the last call to the **Cipher.init()** method. That means the Cipher may be reused to encipher or decipher more data using the same Key and parameters that were specified in the initialization.

Cipher Input and Output Streams

Rather than deal with the complications of buffering enciphered or deciphered data produced by the **Cipher.update()** methods, it may be desirable to use a Java Input/Output Stream type interface. Fortunately, the JCE provides such a mechanism.

The **javax.crypto.CipherInputStream** and **javax.crypto.CipherOutputStream** are based on the Java IO filter streams. This allows them to process data and pass on that data to an underlying stream.

To create a cipher stream, firstly create and initialize a **javax.crypto.Cipher** instance and the underlying stream and then instantiate the required stream type with these two arguments.

For example, the following code fragment will create a **CipherOutputStream** that will encipher its data (using DES) and pass the result to a **ByteArrayOutputStream**. We can access the ciphertext by calling **ByteArrayOutputStream.toByteArray().**

```
Key desKey;
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, desKey);

ByteArrayOutputStream bout = new ByteArrayOutputStream();
CipherOutputStream cout =
                new CipherOutputStream(bout,  cipher);
cout.write("hello world".getBytes());
cout.close();

byte[] cipherText = bout.toByteArray();
```

Once we can encipher and decipher data using a simple stream, interface, we can create much more complicated scenarios. For example, the **OutputStream** could just as easily be a **SocketOutputStream**, or we could construct an **ObjectOutputStream** on top of our cipher stream and encipher Java objects directly.

SealedObject

The **javax.crypto.SealedObject** class provides the mechanism to encipher a Serializable object. This class allows the application to encipher a Java object and then recover the object, all through a simple interface. The **SealedObject** is also serializable, to simplify the transport and storage of the enciphered objects.

A **SealedObject** can be constructed through either serialization or by its constructor. The constructor is used to create a new enciphered object. The constructor's arguments are the object to encipher and the Cipher to use. The provided Cipher instance must be initialized for encryption before the **SealedObject** is created. This means calling a **Cipher.init()** method with **Cipher.ENCRYPT_MODE** as the mode, the required encryption Key and any algorithm parameters.

The following fragment will create a new **SealedObject** containing the enciphered String "**hello world**":

```
Key desKey = ...
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, deskey);

SealedObject so = new SealedObject("hello world", cipher);
```

To recover the original object, the **SealedObject.getObject()** methods may be used. These methods take either a Cipher or Key object. When providing the Cipher parameter, the instance must be initialized in the **Cipher.DECRYPT_MODE** mode, with the appropriate decryption key and the same algorithm parameters as the original Cipher. When providing a Key parameter, the encryption algorithm and algorithm parameters are extracted from the **SealedObject**.

The following fragment will extract a **SealedObject** object from an **ObjectInputStream** and then recover the protected object:

```
ObjectInputStream oin ...
Key desKey = ...

SealedObject so = (SealedObject)oin.readObject();
String plainText = (String)so.getObject(deskey);
```

One important security aspect to note with this class is that it does not use a digital signature to ensure the object has not been tampered with in its serialized form. It is therefore possible that the object could be altered in storage or transport without detection. Fortunately, the JCA provides the java.security.**SignedObject** mechanism, which can be used in conjunction with the **SealedObject** class to avoid this problem. (See "Key Conversion" on page 36 for a discussion on the **SignedObject** class).

Algorithm Parameters

Some cipher algorithms support parameterization. For example, the DES cipher in CBC mode can have an initialization vector as an algorithm parameter and other ciphers may have a selectable block size or round count. The JCE provides support for algorithm-independent initialization via the **java.security.spec.AlgorithmParameterSpec** and **java.security.AlgorithmParameters** classes.

The **java.security.spec.AlgorithmParameterSpec** derived classes can be constructed programatically by an application. The following classes are provided by the JCA/JCE:

java.security.spec	
DSAParameterSpec	Used to specify the parameters used with the DSA algorithm. The parameters consist of the base g, prime p and sub-prime q.

javax.crypto.spec	
DHGenParameterSpec	The set of parameters used for generating Diffie-Hellman parameters for use in Diffie-Hellman key agreement.
DHParameterSpec	The set of parameters used with Diffie-Hellman as specified in PKCS#3.
IvParameterSpec	An initialization vector for use with a feedback cipher. That is an array of bytes of length equal to the block size of the cipher.
RC2ParameterSpec	Parameters for the RC2 algorithm. The parameters are the effective key size and an optional 8-byte initialization vector (only in feedback mode).
RC5ParameterSpec	Parameters for the RC5 algorithm. The parameters are a version number, number of rounds, a word size and an optional initialization vector (only in feedback mode).

Your provider may also include more classes for passing parameters to the algorithms it implements.

The JCA also has mechanisms for dealing with the provider-dependent **AlgorithmParameters**. This class is used as an opaque representation of the parameters for a given algorithm and allows an application to store persistently the parameters used by a Cipher.

There are three situations where an application may encounter an **AlgorithmParameters** instance:

1. **Cipher.getParameters()**

After a Cipher has been initialized, it may have generated a set of parameters (based on supplied and/or default values). The value returned by the **getParameters()** method allows the Cipher to be re-initialized to exactly the same state.

2. **AlgorithmParameters.getInstance()**

Rather than generating the parameters via the Cipher class, it is possible to generate them either based on an encoded format or an **AlgorithmParameterSpec** instance. To do so create an uninitialized instance using the getInstance method and then initialize it by calling the appropriate **init()** method.

3. **AlgorithmParameterGenerator.getParameters()**

Finally, a set of parameters can be generated using the **AlgorithmParameterGenerator**. First, a generator is created for the required algorithm using the **getInstance()** method. Then the generator is initialized by calling one of the **init()** methods, finally to create the instance use the **getParameters** method.

This class provides the concept of algorithm-independent parameter generation, in that the initialization can be based on a "size" and a source of randomness. In this case the "size" value is interpreted differently for each algorithm.

Message Digests

The JCA provides support for the generation of message digests via the **java.security.MessageDigest** class. This class uses the standard factory class design, so to create a **MessageDigest** instance, use the **getInstance()** method with the desired algorithm name and optional provider as parameters.

Once created use the various **update()** methods to process the message data and then finally call the **digest()** method to calculate the final digest. At this point the instance may be reused to calculate a digest for a new message.

```
MessageDigest digest = MessageDigest.getInstance("SHA");

byte[] msg = "The message".getBytes();
digest.update(msg);

byte[] result = digest.digest();
```

Message Authentication Code (MAC)

The **javax.crypto.Mac** API is used to access a "Message Authentication Code" (MAC) algorithm. These algorithms are used to check the integrity of messages upon receipt. There are two classes of MAC algorithms in general, those that are based on message digests (known as HMAC algorithms) and those on encryption algorithms. In both cases a shared secret is required.

A **Mac** is used in the same fashion as a **Cipher**. First, use the factory method **Mac.getInstance()** to get the provider implementation of the required algorithm, then initialize the algorithm with the appropriate key via the **Mac.init()** method. Then, use the **Mac.update()** method to process the message, and finally, use the **Mac.doFinal()** method to calculate the MAC for the message.

To verify the message, follow the same procedure and compare the supplied MAC with the calculated MAC.

Note that it is not necessary to use the **Mac.init()** method to check multiple messages if the shared secret has not changed. The **Mac** will be reset after the call to **Mac.doFinal()** (or a call to **Mac.reset()**).

```
/*
 * on the sender
 */
Mac senderMac = Mac.getInstance("HMAC-SHA1");
senderMac.init(shaMacKey);
byte[] mac = senderMac.doFinal(data);

/*
 * now transmit message and mac to receiver
 */
Mac recMac = Mac.getInstance("HMAC-SHA1");
recMac.init(shaMacKey);
byte[] calcMac = recMac.doFinal(data);

for (int i = 0; i < calcMac.length; i ++)
{
   if (calcMac[i] != mac[i])
   {
      /* bogus MAC! */
      return false;
   }
}

/* all okay */
return true;
```

Authentication

This section describes mechanisms for signing and verifying operations. It contains the following subsections:

> "Digital Signatures" below

> "Object Signing" on page 34

Digital Signatures

The **java.security.Signature** class provides the functionality of a digital signature algorithm. Digital signatures are the digital equivalent of the traditional pen-and paper-signature. They can be used to authenticate the originator of a document, as well as to prove that a person signed the document. Generally, digital signatures are based on public-key encryption, which means that, unlike a MAC, anyone that has access to the public key (and the document) can check the validity of the document.

The Signature interface supports generation and verification of signatures. Once a signature instance has been created using the **Signature.getInstance()** method, it needs to be initialized with the **Signature.initSign()** method for creation of a signature, or **Signature.initVerify()** method for verification of a signature.

Once initialized, the document to be processed should be passed to the signature via the **Signature.update()** methods. Once the entire document has been processed, the **Signature.sign()** method may be called to generate the signature, or the **Signature.verify()** method to verify a supplied signature against a previously generated signature.

After a signature has been generated or verified, the Signature instance is reset to the state it was in after it was last initialized, allowing another signature to be generated or verified using the same key.

One such signature algorithm is "MD5 with RSA", defined in PKCS#1. This algorithm specifies that the document to be signed is passed through the MD5 digest algorithm and then an ASN.1 block containing the digest, along with a digest algorithm identifier, is enciphered using RSA.

To create such a signature:

```
/*
 * Assume this private key is initialized
 */
PrivateKey rsaPrivKey;

/*
 * Create the Signature instance and initialize
 * it for signing with our private key
 */
Signature rsaSig = Signature.getInstance("MD5withRSA");
rsaSig.initSign(rsaPrivKey);

/*
 * Pass in the document data via the update() methods
 */
byte[] document = "The document".getBytes();
rsaSig.update(document);

/*
 * Generate the signature
 */
byte[] signature = rsaSig.sign();
```

To verify the generated signature:

```
/*
 * Assume this public key is initialized
 */
PublicKey rsaPubKey;

/*
 * Create the Signature instance and initialize
 * it for signature verification with the public key
 */
Signature rsaSig = Signature.getInstance("MD5withRSA");
rsaSig.initVerify(rsaPubKey);

/*
 * Pass in the document data via the update() methods
 */
byte[] document = "The document".getBytes();
rsaSig.update(document);

/*
 * Check the generated signature against the supplied
 * signature
 */
if (rsaSig.verify(signature))
{
    // signature okay
}
else
{
    // signature fails
}
```

Object Signing

The **java.security.SignedObject** provides a mechanism for ensuring that a Java object can be authenticated and cannot be tampered with without detection. The mechanism used is similar to the **SealedObject** in that the object to be protected is serialized and then a signature is attached. The **SealedObject** is serializable, so it may be stored or transmitted via the object streams.

To create a **SignedObject**, firstly create an instance of the signature algorithm to use via the **Signature.getInstance()** method, then create the new **SignedObject** instance by providing the object to be signed, the signing key and the Signature instance. Note that there is no need to initialize the **Signature** instance; the **SignedObject** constructor will perform that function.

```
Signature signingEngine = Signature.getInstance(
    "MD5withRSA");
SignedObject so = new SignedObject("hello world",
    privateKey, signingEngine);
```

To verify a **SignedObject**, simply create the **Signature** instance for the required algorithm and then use the **SignedObject.verify()** method with the appropriate **PublicKey**. Again, there is no need to initialize the **Signature** instance.

```
Signature verifyEngine = Signature.getInstance(
    "MD5withRSA");
if (so.verify(publicKey, verifyEngine))
{
    // object okay, extract it
    Object obj = so.getObject();
}
else
{
    // object not authenticated
}
```

Note that this class only provides a mechanism for authentication and verification, it does not provide confidentiality (i.e. encryption). The **SealedObject** may be used for this purpose (see "SealedObject" on page 29). The following example combines these two classes to provide a confidential, authenticated, tamper-proof object:

```
/*
 * sealedObj will contain the signed, enciphered data
 */
SignedObject signedObj = new SignedObject(
    "hello world", privateKey, signingEngine);
SealedObject sealedObj = new SealedObject(
    signedObj, cipher);

/*
 * to verify and recover the original object
 */
SignedObject newObj = sealedObject.getObject(cipher);
if (newObj.verify(publicKey, verificationEngine))
{
    // object verified tampered
    String str = (String)newObj.getObject();
}
else
```

```
{
    // object tampered with!
}
```

Key Management

The JCA/JCE framework manages keys in two forms, a provider-dependent format and a provider-independent format.

The provider-dependent keys will implement either the **java.security.Key** interface (or one of its subclasses) for public-key algorithms, or the **javax.crypto.SecretKey** interface for secret-key algorithms. Provider keys can be generated randomly, via a key agreement algorithm or from their associated provider-independent format.

The provider-independent formats will implement the **java.security.spec.KeySpec** interface. Subclasses of this type exist for both specific key types and for different encoding types. For example, the **java.security.spec.RSAPublicKeySpec** can be used to construct an RSA public key from its modulus and exponent and a **java.security.spec.PKCS8EncodedKeySpec** can be used to construct a private key encoded using PKCS#8.

Each Provider will supply a number of mechanisms that will create the provider-dependent keys or convert the provider-independent keys into provider based keys.

This section contains information on the following:

> "Generating Random Keys" below

> "Key Conversion" on the next page

> "Key Agreement Protocols" on page 38

> "Key Storage" on page 38

> "Certificates" on page 40

Generating Random Keys

The simplest mechanism to create keys for a given provider is to use their random key generators. Random keys are most often generated for use as "session-keys", used for a given dialogue or session and then no longer required. In the case of public-key algorithms, however, they may be generated once and then stored for later use. The JCE framework provides key generation via the following classes:

javax.crypto.KeyGenerator

Generation of symmetric keys (such as DES, IDEA, RC4)

java.security.KeyPairGenerator

Generation of public/private key pairs (such as RSA)

For instance, to create a random 128-bit key for RC4 and initialize a Cipher for encryption with this key:

```
/*
 * Create the key generator for the desired algorithm,
 * and then initialize it for the required key size.
 */
KeyGenerator rc4KeyGen = KeyGenerator.getInstance("RC4");
rc4KenGen.init(128);

/*
```

```
 * Generate the key and then initialize the Cipher
 */
SecretKey rc4Key = rc4KeyGen.generateKey();
Cipher rc4Cipher = Cipher.getInstance("RC4");
rc4Cipher.init(Cipher.ENCRYPT_MODE, rc4Key);
```

Here, the **SecretKey** returned by the **KeyGenerator.generateKey()** method is a provider-dependent key. The returned key can then be used with that provider's algorithms.

Some algorithms have keys that are considered weak, for example with a weak DES key the ciphertext may be the same as the plaintext! Generally, the **KeyGenerator** will not generate those keys, but it is best to check the provider documentation for details on the specific algorithm.

The code to generate a public/private key pair is quite similar:

```
KeyPairGenerator rsaKeyGen = KeyPairGenerator.getInstance("RSA");
rsaKeyGen.initialize(1024);

KeyPair rsaKeyPair = rsaKeyGen.generateKeyPair();
Cipher rsaCipher = Cipher.getInstance("RSA");
rsaCipher.init(Cipher.ENCRYPT_MODE, rsaKeyPair.getPrivate());
```

Key Conversion

Two interfaces are provided to convert between a provider-dependent Key and the provider-independent **KeySpec**: **java.security.KeyFactory** and **javax.crypto.SecretKeyFactory**. The **KeyFactory** class is used for public-key algorithms and the **SecretKeyFactory** class for secret-key algorithms.

An application may choose to store its keys in some way and then recreate the key using a **KeySpec**. For example, the application may contain an embedded RSA public key as two integers; the **RSAPublicKeySpec** along with a **KeyFactory** that can process **RSAPublicKeySpec** instances could then be used to create the provider-dependent key.

Each provider will generally supply a number of **KeyFactory**/**SecretKeyFactory** classes that will accept the various **KeySpec** classes and produce Key instances that may be used with the provider algorithms. These factories are not likely to support all **KeySpec** types, so the provider documentation should provide the details as to what conversions will be accepted.

There are a number of **KeySpec** classes provided by the JCA/JCE:

java.security.spec	
PKCS8EncodedKeySpec	A DER encoding of a private key according to the format specified in the PKCS#8 standard.
X509EncodedKeySpec	A DER encoding of a public or private key, according to the format specified in the X.509 standard.
RSAPublicKeySpec	An RSA public key
RSAPrivateKeySpec	An RSA private key
RSAPrivateCrtKeySpec	An RSA private key, with the Chinese Remainder Theorem (CRT) values

java.security.spec	
DSAPublicKeySpec	A DSA public key
DSAPrivateKeySpec	A DSA private key

javax.crypto.spec	
DESKeySpec	A DES secret key
DESedeKeySpec	A DESede secret key
PBEKeySpec	A user-chosen password that can be used with password base encryption (PBE)
SecretKeySpec	A key that can be represented as a byte array and have no associated parameters. The encoding type is known as RAW.

To convert a **KeySpec** instance into a provider based Key, firstly create a **KeyFactory** or **SecretKeyFactory** of the appropriate type using the **getInstance()** method. Once the instance has been created, use the **KeyFactory.generatePrivate(), KeyFactory.generatePublic()** or **SecretKeyFactory.generateSecret()** method.

In the following example we will create a Key from a **KeySpec** and then recover the **KeySpec** from the Key.

```
/*
 * This is the raw key
 */
byte[] keyBytes = { (byte)0x1, (byte)0x02, (byte)0x03,
    (byte)0x04, (byte)0x05, (byte)0x06, (byte)0x07, (byte)0x08 };

/*
 * Create the provider independent KeySpec
 */
DESKeySpec desKeySpec = new DESKeySpec(keyBytes);

/*
 * Create the KeyFactory to do the Key<->KeySpec translation
 */
SecretKeyFactory keyFact = KeyFactory.getInstance("DES");

/*
 * Create the provider based SecretKey
 */
SecretKey desKey = keyFact.generateSecret(desKeySpec);

/*
 * Convert the provider Key into a generic KeySpec
 */
DESKeySpec desKeySpec2 = keyFact.getKeySpec(desKey,
    DESKeySpec.class);
```

Key Agreement Protocols

Keys may also be generated using the **javax.crypto.KeyAgreement** API. This interface provides the functionality of a key agreement (or key exchange) protocol. For example, a Diffie-Hellman **KeyAgreement** instance would allow two or more parties to generate a shared Diffie-Hellman Key.

To generate the key, it is necessary to call **KeyAgreement.doPhase()** for each party in the exchange with a Key object that represents the current state of the key agreement. The last call to **KeyAgreement.doPhase()** should have the **lastPhase** set to true.

Once all the key agreement phases have been processed, the shared **SecretKey** may be generated by calling the **KeyAgreement.generateSecret()** method.

The **KeyAgreement** API does not define how each of the parties communicates the necessary information for each exchange in the protocol. The required information is passed to the **KeyAgreement.doPhase()** method as a Key. This Key will generally be generated using either a **KeyGenerator** or a **KeyFactory**. The provider documentation will detail the specific steps required for a given protocol.

```
/*
 * Create the KeyAgreement instance for the required
 * protocol and initialize it with our key.  In the
 * case of Diffie-Hellman this would be our private
 * key.
 */
KeyAgreement keyAg = KeyAgreement.getInstance("DH");
keyAg.init(ourKey);

/*
 * Exchange information as per the key exchange
 * protocol.  For DH we would exchange public keys.
 * Note since there is only two parties in this
 * case the return value is not relevant.
 */
keyAg.doPhase(remotePubKey, true);

/*
 * Create the shared secret-key
 */
SecretKey key = keyAg.generateSecret("DES");
```

Key Storage

Once a Key has been generated you may wish to store it for future use. Generally, you'll be saving public/private keys so that you can reuse them at a later date in a key exchange.

The **java.security.KeyStore** API provides one mechanism for management of a number of keys and certificates. There are two types of entries in a **KeyStore**: Key entries and Certificate entries. Key entries are sensitive information, whereas certificates are not.

As Key entries are sensitive, they are therefore are protected by the **KeyStore**. The API allows for a password, or pass phrase, to be attached to each key entry. What the actual implementation does with the password is not defined, although it may be used to encipher the entry. A key entry may either be a **SecretKey**, or a **PrivateKey**. In the case of a **PrivateKey**, the entry is saved along with a Certificate chain, which is the chain of trust. The chain of trust starts with the Certificate containing the corresponding **PublicKey** and ends with a self-signed certificate.

A certificate entry represents a "trusted certificate entry", that is, a Certificate whose identity we trust. This type of entry can be used to authenticate other parties.

To create a **KeyStore** instance, use the **KeyStore.getInstance()** method. This will return an empty **KeyStore** which may then be populated by calling the **KeyStore.load()** method. This method accepts an **InputStream** instance and an optional password. Each individual **KeyStore** will treat these parameters differently, so check the provider documentation for details.

The **Sun** provider supplies a **KeyStore** known as "JKS". This **KeyStore** is used by the **keytool** and **jarsigner** applications.

```
/*
 * Create an instance of the Java Key Store (defined by Sun)
 */
KeyStore keyStore = KeyStore.getInstance("JKS");
```

To add a new entry into the KeyStore, use either **setCertificateEntry()** or one of the **setKeyEntry()** methods. This will add the new entry with the associated alias.

```
char[] myPass;
SecretKey secretKey;

/*
 * Store a SecretKey in the KeyStore, with "mypass"
 * as the password.
 */
keyStore.setKeyEntry("mysecretkey", secretKey,
                                    myPass, null);

/*
 * assume that privateKey contains my PrivateKey
 * and myCert contains a Certificate with the
 * corresponding PublicKey
 */
PrivateKey privateKey;
Certificate myCert;

keyStore.setKeyEntry("myprivatekey", privateKey,
                                    myPass, myCert);
```

To extract an entry, use the **getKey()** method to extract a Key or **getCertificate()** for a Certificate.

```
/*
 * recover the SecretKey
 */
SecretKey key = (SecretKey)keyStore.getKey("mysecretkey",
                                            myPass);

/*
 * recover the PrivateKey
 */
PrivateKey privKey =
      (PrivateKey)keyStore.getKey("myprivatekey", myPass);

/*
 * recover the Certificate (containing the PublicKey)
 * corresponding to our PrivateKey
```

```
    */
    Certificate cert = keyStore.getCertificate("myprivatekey");
```

If the **KeyStore** supports persistence via the **store()** and **load()** methods, the provider documentation will explain what types of Key types may be stored.

Certificates

The JCA framework provides support for generic certificates, as well as X.509v3 certificates. Certificates may be stored using the KeyStore API, or they may be generated from their encoded format (either the PEM or PKCS#7 encoding).

To create a **java.security.cert.Certificate** instance from its encoded format, first create a **java.security.cert.CertificateFactory** instance of the required type (eg X.509). Then use the **generateCertificate()** or **generateCertificates()** methods to convert your **InputStream** into Certificate instances.

```
    CertificateFactory cf =
         CertificateFactory.getInstance("X.509");
    X509Certificate cert =
         (X509Certificate)cf.generateCertificate(inputStream);
```

Two useful methods of the Certificate class are **getPublicKey()** and **verify()**. The first of these allows access to the **PublicKey** of the certificate's owner and the second allows an application to verify that the certificate was signed using the private key that corresponds to the provided public key.

The **java.security.cert.X509Certificate** class, which extends the Certificate class, provides methods to access the other attributes of a X.509 certificate such as the Issuer's distinguished name or its validity period.

The **keytool** application provided with JDK can be used to generate certificates and store them in a **KeyStore**. Check the JDK documentation for information on how to use this application.

Error Handling and Exceptions

The JCA/JCE framework includes a number of specialized exception classes:

java.security	
DigestException	Thrown if an error occurs during the final computation of the digest. Generally this indicates that the output buffer is of insufficient size.
InvalidAlgorithmParameter Exception	Thrown by classes that use **AlgorithmParameters** or **AlgorithmParameterSpec** instances where the supplied instance is not compatible with the algorithm or the supplied parameter was null and the algorithm requires a non-null parameter.
InvalidKeyException	Thrown by the various classes that use Key objects, such as **Signature**, **Mac**, and **Cipher** when the provided Key is not compatible with the given instance.
InvalidParameterException	Only used in the deprecated interfaces in the **Signature** class and the deprecated class **Signer**.

java.security	
KeyStoreException	Thrown by the **KeyStore** class when the object has not been initialized properly.
NoSuchAlgorithmException	Thrown by the **getInstance()** methods when the requested algorithm is not available.
NoSuchProviderException	Thrown by the **getInstance()** methods when the requested provider is not available.
SignatureException	Thrown by the **Signature** class during signature generation or validation if the object has not been initialized correctly or an error occurs in the underlying ciphers.

javax.crypto	
BadPaddingException	Thrown by the **Cipher** class (or classes which use a **Cipher** class to process data) if this cipher is in decryption mode, (un)padding has been requested, and the deciphered data is not bounded by the appropriate padding bytes.
IllegalBlockSizeException	Thrown by the **Cipher** class (or classes which use a **Cipher** class to process data) if this cipher is a block cipher, no padding has been requested (only in encryption mode), and the total input length of the data processed by this cipher is not a multiple of block size
NoSuchPaddingException	Thrown by the **Cipher** class by the **getInstance()** method when a transformation is requested that contains a padding scheme that is not available.
ShortBufferException	Thrown by the **Cipher** class when an output buffer is supplied that is too small to hold the result.

CHAPTER 4: Supported Ciphers

SafeNet ProtectToolkit-J includes support for symmetric block and stream ciphers, as well as support for the asymmetric RSA cipher. The following algorithms are available through the **javax.crypto.Cipher** interface:

Cipher Name	Key Length (bits)	Block Size (bits)	Cipher Modes	Padding
"DES" on page 44	64	64	ECB,CBC	PKCS5Padding, NoPadding
"DESede" on page 47	128,192	64	ECB,CBC	PKCS5Padding, NoPadding
"AES" on page 50	128,182,256	64	ECB,CBC	PKCS5 Padding, NoPadding
"IDEA" on page 52	128	64	ECB,CBC	PKCS5Padding, NoPadding
"CAST128" on page 54	8-128	64	ECB,CBC	PKCS5Padding, NoPadding
"RC2" on page 56	0-1024	64	ECB,CBC	PKCS5Padding, NoPadding
"RC4" on page 59	8-2048	N/A	ECB	NoPadding
PBEWithMD2AndDES ("PBE Ciphers" on page 61)	64	64	N/A	N/A
PBEWithMD5AndDES ("PBE Ciphers" on page 61)	64	64	N/A	N/A
PBEWithMD5AndCAST ("PBE Ciphers" on page 61)	128	128	N/A	N/A
PBEWithSHA1AndCAST ("PBE Ciphers" on page 61)	128	128	N/A	N/A
PBEWithSHA1AndTripleDES ("PBE Ciphers" on page 61)	128	128	N/A	N/A

Cipher Name	Key Length (bits)	Block Size (bits)	Cipher Modes	Padding
"RSA" on page 63	512-4096	variable	ECB	PCKS1Padding, NoPadding, OAEP, OAEPPadding

Here, the Cipher name is the name of the Cipher as known to the JCE. To request a particular algorithm, pass this name to the **Cipher.getInstance()** method. Some algorithms support different key lengths, and the supported key lengths are listed in the table above. The block size is the size of data that is processed by the cipher. During encryption, the amount of data processed must be a multiple of this size, unless padding is employed (see below), and the encrypted output will therefore be a multiple of this size.

Electronic Codebook Mode (ECB) and Cipher Block Chaining (CBC) are defined in *FIPS PUB 81: DES Modes of Operation*. All ciphers will default to ECB mode.

PKCS#5 padding is defined in PKCS#5, and is the standard padding applied to block ciphers with a block size of 64 bits. DES, DESede, IDEA, CAST128 and RC2 all default to "NoPadding". When PKCS5Padding is employed with a block cipher, the input data for encryption can be any length, and will be padded to the appropriate length before encryption.

PKCS#1 padding is defined in PKCS#1, and is the standard padding mechanism for the RSA cipher. When this padding mechanism is used, PKCS#1 padding will be performed on each block encrypted. For public-key encryption PKCS#1 type 1 blocks will be created, and for private-key encryption type 2 blocks will be created. When "NoPadding" is requested, no PKCS#1 packing is applied to the data and the processing is performed as per the X.509 (raw) RSA specification.

Cipher Algorithm Parameters

Currently, SafeNet ProtectToolkit-J does not support algorithm parameters.

Calls to **Cipher.getParameters()** will always return `null`. Neither does the provider include any **java.security.AlgorithmParameters** classes.

DES

This algorithm is a 64-bit block cipher with a 64-bit key. The effective key size is only 56-bit, however, as 8 bits of the key are used for parity. The algorithm described in *FIPS PUB 46-2*.

DES Cipher Initialization

This cipher supports both ECB and CBC modes, and may be used with **NoPadding** or **PKCS5Padding**. To create an instance of this class use the **Cipher.getInstance()** method with "**SAFENET**" as the provider and one of the following strings as the transformation:

> DES

> DES/ECB/NoPadding

> DES/ECB/PKCS5Padding

> DES/CBC/NoPadding

> DES/CBC/PKCS5Padding

Using the "DES" transformation, the Cipher will default to ECB and **NoPadding**.

If the **NoPadding** mode is selected, the input data must be a multiple of 8 bytes; otherwise, the encrypted or decrypted result will be truncated. In **PKCS5Padding**, arbitrary data lengths are accepted; the ciphertext will be padded to a multiple of 8 bytes, as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plaintext is returned.

This Cipher will accept a **javax.crypto.spec.SecretKeySpec** or **au.com.safenet.crypto.provider.CryptokiSecretKey** as the key parameter during initialization.

When the Cipher is initialized in CBC mode, the Initialization Vector (IV) may be specified by passing a **javax.crypto.spec.IvParameterSpec** instance to the **Cipher.init()** method. When decrypting in this mode, a valid IV must be specified in the **Cipher.init()** method. For encryption, however, a random IV will be generated if none is specified (the IV may be retrieved using the **Cipher.getIV()** method).

The IV may be provided as a **java.security.AlgorithmParameters** or a **javax.crypto.spec.IvParameterSpec** instance. If the initialization is done using an **AlgorithmParameters** instance, it must be convertible to an **IvParameterSpec** using the **AlgorithmParameters.getParameterSpec()** method.

This Cipher does not support the **Cipher.getParameters()** method; this method will always return `null`. The only supported parameter for this class is the initialization vector, which may be determined using the **Cipher.getIV()** method.

DES Key

The DES Cipher requires either a **SecretKeySpec** or SafeNet ProtectToolkit-J provider DES Key during initialization.

To create an appropriate **SecretKeySpec**, pass an 8 byte array and the algorithm name "**DES**" to the **SecretKeySpec** constructor. For example:

```
byte[] keyBytes = { 0x01, 0x23, 0x45, 0x67,
                    0x89, 0xAB, 0xCD, 0xEF };
SecretKeySpec desKey = new SecretKeySpec(keyBytes, "DES");
```

Alternatively, a random SafeNet ProtectToolkit-J DES key can be generated randomly using the **KeyGenerator** as described in "Public Keys" on page 79, or from a provider-independent form as described in "Key Specifications" on page 84. The DES key may also be stored in the SafeNet ProtectToolkit-J **KeyStore**, as described in"Key Storage" on page 82 .

The SafeNet ProtectToolkit-J DES key will return the string "**DES**" as its algorithm name, "**RAW**" as its encoding. However, since the key is stored within the hardware, the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as Sensitive. The keys generated in SafeNet ProtectToolkit-J will always be marked as sensitive. It is possible, however, to access any Cryptoki keys stored on the device, and it is possible that the attributes of these keys have been modified.

DES KeyGenerator

The DES **KeyGenerator** is used to generate random DES keys. The generated key will be a hardware key that has the **Cryptoki CKA_EXTRACTABLE** and **CKA_SENSITIVE** attributes set. Since these keys are marked as sensitive, their **getEncoded()** method will return `null`.

During initialization, the strength and random parameters are ignored, as all keys are 64-bits and the hardware includes a cryptographically-secure random source.

Keys generated using the **KeyGenerator** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

DES SecretKeyFactory

The DES **SecretKeyFactory** is used to construct SafeNet ProtectToolkit-J keys from their provider-independent form. The provider independent form of the DES key is the **javax.crypto.spec.DESKeySpec** class.

Keys generated using the **SecretKeyFactory** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

For example, to create the provider based key from its provider-independent form:

```
byte[] keyBytes = { 0x01, 0x23, 0x45, 0x67,
                    0x89, 0xAB, 0xCD, 0xEF };
DESKeySpec desKeySpec = new DESKeySpec(keyBytes);
SecretKeyFactory desKeyFact =
            SecretKeyFactory.getInstance("DES", "SAFENET");
SecretKey desKey = desKeyFact.generateSecret(desKeySpec);
```

DES Example Code

The following example code will create a random DES key, then create a DES cipher in CBC mode with **PKCS5Padding**. Next, it initializes the cipher for encryption using the newly-created key. We then save the initialization vector and encrypt the string "**hello world**".

To perform the decryption, we re-initialize the cipher in decrypt mode, with the same key and the initialization vector that was created during encryption.

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES",
                                        "SAFENET");
Key desKey = keyGen.generateKey();
```

```
Cipher desCipher = Cipher.getInstance("DES/CBC/PKCS5Padding",
                                      "SAFENET");
desCipher.init(Cipher.ENCRYPT_MODE, desKey);
byte[] iv = desCipher.getIV();
byte[] cipherText = desCipher.doFinal(
                                "hello world".getBytes());
desCipher.init(Cipher.DECRYPT_MODE, desKey,
                      new IvParameterSpec(iv));
byte[] plainText = desCipher.doFinal(cipherText);
```

DESede

This algorithm, known as triple-DES, is a 64-bit block cipher with a 192-bit key, although 24 bits of the key are parity bits. This algorithm works by splitting the 192-bit key into three 64-bit keys and then applying the basic DES cipher, first in the encrypt mode, second in the decrypt mode, and finally in the encrypt mode. The algorithm is described in *ANSI X9.17*. It is also possible to use a double-length key (128 bits), in this case the first key is reused as the final key.

DESede Cipher Initialization

This cipher supports both ECB and CBC modes, and may be used with NoPadding or PKCS5Padding. To create an instance of this class, use the **Cipher.getInstance()** method with "**SAFENET**" as the provider and one of the following strings as the transformation:

> DESede

> DESede/ECB/NoPadding

> DESede/ECB/PKCS5Padding

> DESede/CBC/NoPadding

> DESede/CBC/PKCS5Padding

Using the "**DESede**" transformation, the Cipher will default to ECB and **NoPadding**.

If the **NoPadding** mode is selected, the input data must be a multiple of 8 bytes; otherwise, the encrypted or decrypted result will be truncated. In **PKCS5Padding**, arbitrary data lengths are accepted; the ciphertext will be padded to a multiple of 8 bytes, as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plaintext is returned.

This Cipher will accept a **javax.crypto.spec.SecretKeySpec** or **au.com.safenet.crypto.provider.CryptokiSecretKey** as the key parameter during initialization.

When the Cipher is initialized in CBC mode, the Initialization Vector (IV) may be specified by passing a **javax.crypto.spec.IvParameterSpec** instance to the **Cipher.init()** method. When decrypting in this mode, a valid IV must be specified in the **Cipher.init()** method. For encryption, however, a random IV will be generated if none is specified (the IV may be retrieved using the **Cipher.getIV()** method).

The IV may be provided as a **java.security.AlgorithmParameters** or a **javax.crypto.spec.IvParameterSpec** instance. If the initialization is done using an **AlgorithmParameters** instance, it must be convertible to an **IvParameterSpec** using the **AlgorithmParameters.getParameterSpec()** method.

This Cipher does not support the **Cipher.getParameters()** method; this method will always return `null`. The only supported parameter for this class is the initialization vector, which may be determined using the **Cipher.getIV()** method.

DESede Key

The DESede Cipher requires either a **SecretKeySpec** or SafeNet ProtectToolkit-J provider DESede Key during initialization. The DESede key may be either a double- or triple-length key.

To create an appropriate **SecretKeySpec**, pass a 16 or 24-byte array and the algorithm name "**DESede**" to the **SecretKeySpec** constructor. For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
SecretKeySpec desEdeKey = new SecretKeySpec(keyBytes,
                                            "DESede");
```

Alternatively, a random SafeNet ProtectToolkit-J DESede key can be generated using the **KeyGenerator** as described in section "Public Keys" on page 79, or a provider-independent form as described in section "Key Specifications" on page 84. The DESede key may also be stored in the SafeNet ProtectToolkit-J **KeyStore**, as described in"Key Storage" on page 82 .

The SafeNet ProtectToolkit-J DESede key will return the string "**DESede**" as its algorithm name, and "**RAW**" as its encoding. However, since the key is stored within the hardware, the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as Sensitive. The keys generated in SafeNet ProtectToolkit-J will always be marked as sensitive. It is possible, however, to access any Cryptoki keys stored on the device, and it is possible that the attributes of these keys have been modified.

DESede KeyGenerator

The DESede **KeyGenerator** is used to generate random DESede double or triple length keys. The generated key will be a hardware key that has the **Cryptoki CKA_EXTRACTABLE** and **CKA_SENSITIVE** attributes set. Since these keys are marked as Sensitive, their **getEncoded()** method will return `null`.

During initialization, the strength parameter may be 128 to specify a double length key or 196 to specify a triple-length key. If no strength is specified, a triple-length key will be generated. The random parameter is ignored as the hardware includes a cryptographically-secure random source.

Keys generated using the **KeyGenerator** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

DESede SecretKeyFactory

The DESede **SecretKeyFactory** is used to construct SafeNet ProtectToolkit-J keys from their provider-independent form. The provider-independent form of the DESede key is the **javax.crypto.spec.DESedeKeySpec** class.

Keys generated using the **SecretKeyFactory** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

For example, to create the provider based key from its provider independent form (in this case we are generating a triple-length key; specify 16 bytes for a double-length key):

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF,
                    0x39, 0xDF, 0x28, 0x94,
                    0x11, 0x93, 0x55, 0x67,
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
DESedeKeySpec desEdeKeySpec = new DESedeKeySpec(keyBytes);
SecretKeyFactory desEdeKeyFact =
```

```
        SecretKeyFactory.getInstance("DESede", "SAFENET");
SecretKey desEdeKey =
        desEdeKeyFact.generateSecret(desEdeKeySpec);
```

DESede Example Code

See for the simple DES example. To convert the example to use DESede, use "**DESede**" in place of "**DES**".

AES

This algorithm is an implementation of AES, which is a 64 bit block cipher with a variable length key 128, 192 or 256 bits long.

AES Cipher Initialization

This cipher supports both ECB and CBC modes, and may be used with **NoPadding** or **PKCS5Padding**. To create an instance of this class use the **Cipher.getInstance()**method with "**SAFENET**" as the provider and one of the following strings as the transformation:

> AES

> AES/ECB/NoPadding

> AES/CBC/NoPadding

> AES/CBC/PKCS5Padding

Using the "**AES**" transformation, the Cipher will default to ECB and **NoPadding**.

If the **NoPadding** mode is selected, the input data must be a multiple of 8 bytes; otherwise, the encrypted or decrypted result will be truncated. In **PKCS5Padding**, arbitrary data lengths are accepted; the ciphertext will be padded to a multiple of 8 bytes, as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plaintext is returned.

This Cipher will accept a **javax.crypto.spec.SecretKeySpec** or **au.com.safenet.crypto.provider.CryptokiSecretKey** as the key parameter during initialization.

When the Cipher is initialized in CBC mode, the Initialization Vector (IV) may be specified by passing a **javax.crypto.spec.IvParameterSpec** instance to the **Cipher.init()** method. When decrypting in this mode, a valid IV must be specified in the **Cipher.init()** method. For encryption, however, a random IV will be generated if none is specified (the IV may be retrieved using the **Cipher.getIV()** method).

The IV may be provided as a **java.security.AlgorithmParameters** or a **javax.crypto.spec.IvParameterSpec** instance. If the initialization is done using an **AlgorithmParameters** instance, it must be convertible to an **IvParameterSpec** using the **AlgorithmParameters.getParameterSpec()** method.

This Cipher does not support the **Cipher.getParameters()** method; this method will always return `null`. The only supported parameter for this class is the initialization vector, which may be determined using the **Cipher.getIV()** method.

AES Key

The AES Cipher requires either a **SecretKeySpec** or SafeNet ProtectToolkit-J provider AES Key during initialization. AES keys can be 128, 192, or 256 bits long.

To create an appropriate **SecretKeySpec**, pass a 16, 24 or 32 byte array and the algorithm name "**AES**" to the **SecretKeySpec** constructor.

For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xB6, 0xDC, 0x34,
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
SecretKeySpec aesKey = new SecretKeySpec(keyBytes, "AES");
```

Alternatively, a random SafeNet ProtectToolkit-J AES key can be generated using the **KeyGenerator** as described in "Key Generation" on page 78, or a provider-independent form. The AES key may also be stored in the SafeNet ProtectToolkit-J **KeyStore**, as described in "Key Storage" on page 82.

The SafeNet ProtectToolkit-J AES key will return the string "AES" as its algorithm name, "RAW" as its encoding. However, since the key is stored within the hardware the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as Sensitive. The keys generated in SafeNet ProtectToolkit-J will always be marked as sensitive. It is possible, however, to access any Cryptoki keys stored on the device, and it is possible that the attributes of these keys have been modified.

AES KeyGenerator

The AES KeyGenerator is used to generate random AES keys. The generated key will be a hardware key that has the **Cryptoki CKA_EXTRACTABLE** and **CKA_SENSITIVE** attributes set. Since these keys are marked as sensitive their **getEncoded()** method will return null.

During initialization, the strength parameter may only be 128, 192, or 256 bits, with the default size being 128 bits. The random parameter is ignored as the hardware includes a cryptographically-secure random source.

Keys generated using the **KeyGenerator** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

AES SecretKeyFactory

The AES **SecretKeyFactory** is used to construct SafeNet ProtectToolkit-J keys from their provider-independent form. The provider-independent form of the AES key is the **au.com.safenet.crypto.spec.AESKeySpec class**.

Keys generated using the **SecretKeyFactory** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

For example, to create the provider-based key from its provider-independent form:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
AESKeySpec ideaKeySpec = new AESKeySpec(keyBytes);
SecretKeyFactory aesKeyFact =
        SecretKeyFactory.getInstance("AES", "SAFENET");
SecretKey aesKey = aesKeyFact.generateSecret(aesKeySpec);
```

AES Example Code

See "DES" on page 44 for the simple DES example. To convert the example to use AES, use "**AES**" in place of "**DES**".

IDEA

This algorithm is a 64-bit block cipher with a 128-bit key. The last patents on this algorithm expired in 2012, and IDEA is now free for all uses.

IDEA Cipher Initialization

This cipher supports both ECB and CBC modes, and may be used with **NoPadding** or **PKCS5Padding**. To create an instance of this class, use the **Cipher.getInstance()** method with "**SAFENET**" as the provider and one of the following strings as the transformation:

> IDEA

> IDEA/ECB/NoPadding

> IDEA/ECB/PKCS5Padding

> IDEA/CBC/NoPadding

> IDEA/CBC/PKCS5Padding

Using the "**IDEA**" transformation the Cipher will default to ECB and **NoPadding**.

If the **NoPadding** mode is selected, the input data must be a multiple of 8 bytes; otherwise, the encrypted or decrypted result will be truncated. In **PKCS5Padding**, arbitrary data lengths are accepted; the ciphertext will be padded to a multiple of 8 bytes, as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plaintext is returned.

This Cipher will accept a **javax.crypto.spec.SecretKeySpec** or **au.com.safenet.crypto.provider.CryptokiSecretKey** as the key parameter during initialization.

When the Cipher is initialized in CBC mode, the Initialization Vector (IV) may be specified by passing a **javax.crypto.spec.IvParameterSpec** instance to the **Cipher.init()** method. When decrypting in this mode, a valid IV must be specified in the **Cipher.init()** method. For encryption, however, a random IV will be generated if none is specified (the IV may be retrieved using the **Cipher.getIV()** method).

The IV may be provided as a **java.security.AlgorithmParameters** or a **javax.crypto.spec.IvParameterSpec** instance. If the initialization is done using an **AlgorithmParameters** instance, it must be convertible to an **IvParameterSpec** using the **AlgorithmParameters.getParameterSpec()** method.

This Cipher does not support the **Cipher.getParameters()** method; this method will always return `null`. The only supported parameter for this class is the initialization vector, which may be determined using the **Cipher.getIV()** method.

IDEA Key

The IDEA Cipher requires either a **SecretKeySpec** or SafeNet ProtectToolkit-J provider IDEA Key during initialization. The IDEA key is always 128 bits long.

To create an appropriate **SecretKeySpec**, pass a 16 byte array and the algorithm name "**IDEA**" to the **SecretKeySpec** constructor. For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xB6, 0xDC, 0x34,
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
SecretKeySpec ideaKey = new SecretKeySpec(keyBytes, "IDEA");
```

Alternatively, a random SafeNet ProtectToolkit-J IDEA key can be generated using the **KeyGenerator** as described in section "Public Keys" on page 79, or from a provider-independent form as described in section "Key Specifications" on page 84. The IDEA key may also be stored in the SafeNet ProtectToolkit-J **KeyStore** as described in"Key Storage" on page 82 .

The SafeNet ProtectToolkit-J IDEA key will return the string "**IDEA**" as its algorithm name, "**RAW**" as its encoding. However, since the key is stored within the hardware, the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as Sensitive. The keys generated in SafeNet ProtectToolkit-J will always be marked as sensitive. It is possible, however, to access any Cryptoki keys stored on the device, and it is possible that the attributes of these keys have been modified.

IDEA KeyGenerator

The IDEA **KeyGenerator** is used to generate random IDEA keys. The generated key will be a hardware key that has the **Cryptoki CKA_EXTRACTABLE** and **CKA_SENSITIVE** attributes set. Since these keys are marked as sensitive their **getEncoded()** method will return `null`.

During initialization the strength and random parameters are ignored, as all keys are 128-bits and the hardware includes a cryptographically-secure random source.

Keys generated using the **KeyGenerator** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

IDEA SecretKeyFactory

The IDEA **SecretKeyFactory** is used to construct SafeNet ProtectToolkit-J keys from their provider-independent form. The provider-independent form of the IDEA key is the **au.com.safenet.crypto.spec.IDEAKeySpec** class.

Keys generated using the **SecretKeyFactory** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

For example, to create the provider-based key from its provider-independent form:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
IDEAKeySpec ideaKeySpec = new IDEAKeySpec(keyBytes);
SecretKeyFactory ideaKeyFact =
        SecretKeyFactory.getInstance("IDEA", "SAFENET");
SecretKey ideaKey = ideaKeyFact.generateSecret(ideaKeySpec);
```

IDEA Example Code

See "DES" on page 44 for the simple DES example. To convert the example to use IDEA, use "**IDEA**" in place of "**DES**".

CAST128

This algorithm is an implementation of CAST-128, a 64-bit block cipher with a variable length key from 8 to 128 bits. The algorithm is described in *RFC-2144*.

CAST128 Cipher Initialization

This cipher supports both ECB and CBC modes, and may be used with **NoPadding** or **PKCS5Padding**. To create an instance of this class, use the **Cipher.getInstance()** method with "**SAFENET**" as the provider and one of the following strings as the transformation:

> CAST128

> CAST128/ECB/NoPadding

> CAST128/ECB/PKCS5Padding

> CAST128/CBC/NoPadding

> CAST128/CBC/PKCS5Padding

Using the "**CAST128**" transformation, the Cipher will default to ECB and **NoPadding**.

If the **NoPadding** mode is selected, the input data must be a multiple of 8 bytes; otherwise, the encrypted or decrypted result will be truncated. In **PKCS5Padding**, arbitrary data lengths are accepted; the ciphertext will be padded to a multiple of 8 bytes, as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plaintext is returned.

This Cipher will accept a **javax.crypto.spec.SecretKeySpec** or **au.com.safenet.crypto.provider.CryptokiSecretKey** as the key parameter during initialization.

When the Cipher is initialized in CBC mode, the Initialization Vector (IV) may be specified by passing a **javax.crypto.spec.IvParameterSpec** instance to the **Cipher.init()** method. When decrypting in this mode, a valid IV must be specified in the **Cipher.init()** method. For encryption, however, a random IV will be generated if none is specified (the IV may be retrieved using the **Cipher.getIV()** method).

The IV may be provided as a **java.security.AlgorithmParameters** or a **javax.crypto.spec.IvParameterSpec** instance. If the initialization is done using an **AlgorithmParameters** instance, it must be convertible to an **IvParameterSpec** using the **AlgorithmParameters.getParameterSpec()** method.

This Cipher does not support the **Cipher.getParameters()** method; this method will always return `null`. The only supported parameter for this class is the initialization vector, which may be determined using the **Cipher.getIV()** method.

CAST128 Key

The CAST128 Cipher requires either a **SecretKeySpec** or SafeNet ProtectToolkit-J provider CAST128 Key during initialization. The CAST128 key may be any length of 8 to 128 bits.

To create an appropriate **SecretKeySpec**, pass an array of up to 16 bytes and the algorithm name "**CAST128**" to the **SecretKeySpec** constructor. For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDF, 0x28, 0x94,
                    0x11, 0x93, 0x55, 0x67,
```

```
                     0x39, 0xAC, 0xCD, 0xFF };
SecretKeySpec castKey = new SecretKeySpec(keyBytes,
                                        "CAST128");
```

Alternatively, a random SafeNet ProtectToolkit-J CAST128 key can be generated using the **KeyGenerator** as described in "Key Generation" on page 78, or, a provider-independent form. The CAST128 key may also be stored in the SafeNet ProtectToolkit-J **KeyStore** as described in "Key Storage" on page 82.

The SafeNet ProtectToolkit-J CAST128 key will return the string "**CAST128**" as its algorithm name, "**RAW**" as its encoding. However, since the key is stored within the hardware, the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as Sensitive. The keys generated in SafeNet ProtectToolkit-J will always be marked as sensitive. It is possible, however, to access any Cryptoki keys stored on the device, and it is possible that the attributes of these keys have been modified.

CAST128 KeyGenerator

The CAST128 **KeyGenerator** is used to generate random CAST128 keys. The generated key will be a hardware key that has the **Cryptoki CKA_EXTRACTABLE** and **CKA_SENSITIVE** attributes set. Since these keys are marked as sensitive, their **getEncoded()** method will return `null`.

During initialization, the strength parameter may be any length from 8 to 128. The default key size is 128 bits. The random parameter is ignored as the hardware includes a cryptographically-secure random source.

Keys generated using the **KeyGenerator** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

CAST128 SecretKeyFactory

The CAST128 **SecretKeyFactory** is used to construct SafeNet ProtectToolkit-J keys from their provider-independent form. The provider-independent form of the CAST128 key is the **au.com.safenet.crypto.spec.CASTKeySpec** class.

Keys generated using the **SecretKeyFactory** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

For example, to create the provider-based key from its provider-independent form:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
CAST128KeySpec castKeySpec = new CAST128KeySpec(keyBytes);
SecretKeyFactory castKeyFact =
        SecretKeyFactory.getInstance("CAST128", "SAFENET");
SecretKey castKey=castKeyFact.generateSecret(castEdeKeySpec);
```

CAST128 Example Code

See "DES Example Code" on page 45 for the simple DES example. To convert the example to use CAST128, use "**CAST128**" in place of "**DES**".

RC2

This algorithm is a 64-bit block cipher with a variable-length key usually 40-bit or 128-bit. RC2 was designed by Ron Rivest and is a trademark of RSA Data Security. For more information on this algorithm, see *RFC-2268*.

RC2 Cipher Initialization

This cipher supports both ECB and CBC modes, and may be used with **NoPadding** or **PKCS5Padding**. To create an instance of this class, use the **Cipher.getInstance()** method with "**SAFENET**" as the provider and one of the following strings as the transformation:

> RC2

> RC2/ECB/NoPadding

> RC2/ECB/PKCS5Padding

> RC2/CBC/NoPadding

> RC2/CBC/PKCS5Padding

Using the "**RC2**" transformation, the Cipher will default to ECB and **NoPadding**.

If the **NoPadding** mode is selected, the input data must be a multiple of 8 bytes; otherwise, the encrypted or decrypted result will be truncated. In **PKCS5Padding**, arbitrary data lengths are accepted; the ciphertext will be padded to a multiple of 8 bytes, as described in PKCS#5. The decryption process will remove the padding from the data so that the correct plaintext is returned.

This Cipher will accept a **javax.crypto.spec.SecretKeySpec** or **au.com.safenet.crypto.provider.CryptokiSecretKey** as the key parameter during initialization.

The RC2 Cipher may also be initialized with an instance of the **javax.crypto.spec.RC2ParameterSpec** class. With this class it is possible to supply an initialization vector and an effective key size. If the Cipher is not initialized in this way, the effective key size will default to 128.

The IV may be provided as a **java.security.AlgorithmParameters** or a **javax.crypto.spec.IvParameterSpec** instance. If the initialization is done using an **AlgorithmParameters** instance, it must be convertible to an **IvParameterSpec** using the **AlgorithmParameters.getParameterSpec()** method.

This Cipher does not support the **Cipher.getParameters()** method; this method will always return `null`. The only supported parameter for this class is the initialization vector, which may be determined using the **Cipher.getIV()** method.

RC2 Key

The RC2 Cipher requires either a **SecretKeySpec** or SafeNet ProtectToolkit-J provider RC2 Key during initialization. The RC2 key may be any length of 8 to 1024 bits.

To create an appropriate **SecretKeySpec**, pass an array of up to 128 bytes and the algorithm name "**RC2**" to the **SecretKeySpec** constructor. For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDF, 0x28, 0x94,
                    0x11, 0x93, 0x55, 0x67,
```

```
                          0x39, 0xAC, 0xCD, 0xFF };
SecretKeySpec rc2Key = new SecretKeySpec(keyBytes, "RC2");
```

Alternatively, a random SafeNet ProtectToolkit-J RC2 key can be generated using the **KeyGenerator** as described in section "Public Keys" on page 79, or a provider-independent form as described in section "Key Specifications" on page 84. The RC2 key may also be stored in the SafeNet ProtectToolkit-J **KeyStore**, as described in"Key Storage" on page 82.

The SafeNet ProtectToolkit-J RC2 key will return the string "**RC2**" as its algorithm name, "**RAW**" as its encoding. However, since the key is stored within the hardware, the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as Sensitive. The keys generated in SafeNet ProtectToolkit-J will always be marked as sensitive. It is possible, however, to access any Cryptoki keys stored on the device, and it is possible that the attributes of these keys have been modified.

RC2 KeyGenerator

The RC2 **KeyGenerator** is used to generate random RC2 keys. The generated key will be a hardware key that has the **Cryptoki CKA_EXTRACTABLE** and **CKA_SENSITIVE** attributes set. Since these keys are marked as sensitive, their **getEncoded()** method will return null.

During initialization, the strength parameter may be any multiple of 8 up to 1024 inclusive. The default key size is 128 bits. The random parameter is ignored as the hardware includes a cryptographically-secure random source.

Keys generated using the **KeyGenerator** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

RC2 SecretKeyFactory

The RC2 **SecretKeyFactory** is used to construct SafeNet ProtectToolkit-J keys from their provider-independent form. The provider-independent form of the RC2 key is the **au.com.safenet.crypto.spec.RC2KeySpec** class.

Keys generated using the **SecretKeyFactory** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

For example, to create the provider based key from its provider-independent form:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
RC2KeySpec rc2KeySpec = new RC2KeySpec(keyBytes);
SecretKeyFactory rc2KeyFact =
        SecretKeyFactory.getInstance("RC2", "SAFENET");
SecretKey rc2Key = rc2KeyFact.generateSecret(castEdeKeySpec);
```

RC2 Example Code

See"DES" on page 44 for the simple DES example. To convert the example to use RC2, use "**RC2**" in place of "**DES**".

Replace the **IvParameterSpec** call with the **RC2ParameterSpec** call, as illustrated in the following code example:

```
KeyGenerator keyGen = KeyGenerator.getInstance("RC2","SAFENET");
Key rcKey = keyGen.generateKey();
Cipher rc2Cipher = Cipher.getInstance("RC2/CBC/PKCS5Padding","SAFENET");
rc2Cipher.init(Cipher.ENCRYPT_MODE, rcKey);
byte[] iv = rc2Cipher.getIV();
byte[] cipherText = rc2Cipher.doFinal("hello world".getBytes());
rc2Cipher.init(Cipher.DECRYPT_MODE, rcKey,new RC2ParameterSpec(iv));
byte[] plainText = rc2Cipher.doFinal(cipherText);
```

RC4

This algorithm is a stream cipher with a variable length key, usually 40-bit or 128-bit. RC4 is a trademark of RSA Data Security. A description of the algorithm may be found in *Applied Cryptography* by Bruce Schneier.

RC4 Cipher Initialization

Since the RC4 Cipher is a stream cipher, it always operates in the same mode, which may be specified by the transformations "**RC4**" or "**RC4/ECB/NoPadding**". To create an instance of this class, use the **Cipher.getInstance()** method with "**SAFENET**" as the provider and one of the valid transformation strings.

The size of the output of this cipher will always be the same as that of the input.

This Cipher will accept a **javax.crypto.spec.SecretKeySpec** or **au.com.safenet.crypto.provider.CryptokiSecretKey** as the key parameter during initialization.

This Cipher does not support initialization with algorithm parameters, and so the **Cipher.getParameters()** method will always return `null`.

RC4 Key

The RC4 Cipher requires either a **SecretKeySpec** or SafeNet ProtectToolkit-J provider RC4 Key during initialization. The RC4 key may be any length of 8 to 2048 bits.

To create an appropriate **SecretKeySpec**, pass an array of up to 256 bytes and the algorithm name "**RC4**" to the **SecretKeySpec** constructor. For example:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDF, 0x28, 0x94,
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
SecretKeySpec desKey = new SecretKeySpec(keyBytes, "RC4");
```

Alternatively, a random SafeNet ProtectToolkit-J RC4 key can be generated using the **KeyGenerator**, as described in section "Public Keys" on page 79, or a provider-independent form as described in section "Key Specifications" on page 84. The RC4 key may also be stored in the SafeNet ProtectToolkit-J **KeyStore**, as described in"Key Storage" on page 82 .

The SafeNet ProtectToolkit-J RC4 key will return the string "**RC4**" as its algorithm name, "**RAW**" as its encoding. However, since the key is stored within the hardware, the actual key encoding may not be available.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as Sensitive. The keys generated in SafeNet ProtectToolkit-J will always be marked as sensitive. It is possible, however, to access any Cryptoki keys stored on the device, and it is possible that the attributes of these keys have been modified.

RC4 KeyGenerator

The RC4 **KeyGenerator** is used to generate random RC4 keys. The generated key will be a hardware key that has the **Cryptoki CKA_EXTRACTABLE** and **CKA_SENSITIVE** attributes set. Since these keys are marked as sensitive, their **getEncoded()** method will return `null`.

During initialization, the strength parameter may be any length from 8 to 2048. The default key size is 128 bits. The random parameter is ignored as the hardware includes a cryptographically-secure random source.

Keys generated using the **KeyGenerator** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

RC4 SecretKeyFactory

The RC4 **SecretKeyFactory** is used to construct SafeNet ProtectToolkit-J keys from their provider-independent form. The provider-independent form of the RC4 key is the **au.com.safenet.crypto.spec.RC4KeySpec** class.

Keys generated using the **SecretKeyFactory** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

For example, to create the provider-based key from its provider-independent form:

```
byte[] keyBytes = { 0x41, 0x22, 0x35, 0x17,
                    0x39, 0xDB, 0xDC, 0xEF
                    0x11, 0x93, 0x55, 0x67,
                    0x39, 0xAC, 0xCD, 0xFF };
RC4KeySpec castKeySpec = new RC4KeySpec(keyBytes);
SecretKeyFactory castKeyFact =
        SecretKeyFactory.getInstance("RC4", "SAFENET");
SecretKey castKey=castKeyFact.generateSecret(castEdeKeySpec);
```

RC4 Example Code

The following example code will create a random RC4 key, then create a RC4 cipher. Next, it initializes the cipher for encryption using the newly-created key. We then save the initialization vector and encrypt the string "**hello world**".

To perform the decryption, we simply re-initialize the cipher in decrypt mode, with the same key. In this case there is no need to process the initialization vector, as there is none with the RC4 algorithm.

```
KeyGenerator keyGen = KeyGenerator.getInstance("RC4",
                                            "SAFENET");
Key rc4Key = keyGen.generateKey();
Cipher rc4Cipher = Cipher.getInstance("RC4", "SAFENET");
rc4Cipher.init(Cipher.ENCRYPT_MODE, rc4Key);
byte[] cipherText = rc4Cipher.doFinal(
                            "hello world".getBytes());
rc4Cipher.init(Cipher.DECRYPT_MODE, rc4Key);
byte[] plainText = desCipher.doFinal(cipherText);
```

PBE Ciphers

A PBE Cipher is a password based cipher. It allows keying of a cipher based on a user supplied password. PKCS#5 is the standard which defines the generic PBE algorithm used by all PBE algorithms except for the **PBEWithSHA1AndTripleDES** algorithm, which uses PKCS#12 (see *PKCS #12: Personal Information Exchange Syntax Standard*). A particular PBE implementation will combine a message digest algorithm (such as MD5) with a symmetric encryption algorithm (DES, for example).

SafeNet ProtectToolkit-J includes five password-based Ciphers. They are essentially identical, with the password-generation differences below:

> **PBEWithMD2AndDES** - uses MD2 in password generation

> **PBEWithMD5AndDES** - uses MD5 in password generation

> **PBEWithMD5AndCAST** - uses MD5 in password generation

> **PBEWithSHA1AndCAST** - uses SHA1 in password generation

> **PBEWithSHA1AndTripleDES** - uses SHA1 in password generation

As the names suggest, these ciphers use either DES, CAST, or TripleDES as their encryption algorithm, and are therefore 64-bit block ciphers. They are all operated with the block cipher in CBC mode; however, the initialization vector is determined from the password, so there is no need to supply its value.

PBE Cipher Initialization

A PBE Cipher will always operate with the underlying Cipher in a specific mode. For SafeNet ProtectToolkit-J, the DES Cipher will operate in CBC mode with **PCKS5Padding**. Thus, the only valid transformations that may be passed to the **Cipher.getInstance()** method are **PBEWithMD2AndDES**, **PBEWithMD5AndDES** , **PBEWithMD5AndCAST**, **PBEWithSHA1AndCAST**, or **PBEWithSHA1AndTripleDES**.

This Cipher will only accept a SafeNet ProtectToolkit-J provider PBE key as the key parameter during initialization. To create such a Key, use the PBE **SecretKeyFactory** described below.

This Cipher also requires initialization with a valid **PBEParameterSpec** instance, (or an **AlgorithmParameters** instance that can be converted to the generic form via the **getParameterSpec()** method). This parameters instance is used to supply the salt and iteration count parameters to the PBE Cipher. This is a required parameter, there are no defaults and so the **Cipher.getParameters()** method, this will always return `null`.

PBE Key

The PBE Cipher instances require initialization with a SafeNet ProtectToolkit-J provider PBE key. Instances of this type may be created using the PBE **SecretKeyFactory**. The PBE **SecretKeyFactory** is used to construct SafeNet ProtectToolkit-J keys from their provider-independent form. The provider independent form of the PBE key is the **javax.crypto.spec.PBEKeySpec** class.

For example, to create the provider based key from its provider independent form:

```
PBEKeySpec pbeKS = new PBEKeySpec("password".toCharArray())
SecretKeyFactory pbeKF = SecretKeyFactory.getInstance("PBE",
                                            "SAFENET");
Key key = pbeKF.generateSecret(pbeKS);
```

The SafeNet ProtectToolkit-J PBE key will return the string "**PBE**" as its algorithm name, "**RAW**" as its encoding. However, this key class does not support encoding and so will return `null` from the **getEncoded()** method.

PBE Example Code

The following example code will create a PBE key with the string "**password**", convert this into a SafeNet ProtectToolkit-J PBE key, then create a PBE cipher. Next it initializes the cipher for encryption using the newly-created key and the PBE parameters with a salt of "**salt**" and an iteration count of 5. Finally we encrypt the string "**hello world**".

To perform the decryption, we simply re-initialize the cipher in decrypt mode, with the same key and parameters.

```
PBEKeySpec pbeKS = new PBEKeySpec("password".toCharArray())
SecretKeyFactory pbeKF = SecretKeyFactory.getInstance("PBE",
                                                "SAFENET");
Key pbeKey = pbeKF.generateSecret(pbeKS);
PBEParameterSpec pbeParams =
                new PBEParameterSpec("salt".getBytes, 5);
Cipher pbeCipher = Cipher.getInstance("PBEWithMD5andDES",
                                                "SAFENET");
pbeCipher.init(Cipher.ENCRYPT_MODE, pbeKey, pbeParams);
byte[] cipherText = pbeCipher.doFinal(
                        "hello world".getBytes());
pbeCipher.init(Cipher.DECRYPT_MODE, pbeKey, pbeParams);
byte[] plainText = pbeCipher.doFinal(cipherText);
```

RSA

This algorithm is a block cipher with a variable-length key, whose block size is equal to the key size. RSA is patented in the United States by RSA Data Security. The RSA cipher will operate in one of five modes, depending on the padding requested. If "**PKCS1Padding**" is requested, the processing is performed as described in PKCS#1. If "**NoPadding**" is requested, the processing is performed as specified in X.509 for raw RSA.

> **NOTE**   Currently the RSA Cipher only supports encryption or decryption of a single block. Any attempt to pass more data than a single block will result in a RuntimeException.

RSA Cipher Initialization

This cipher supports both only ECB mode, and may be used with **NoPadding** or **PKCS1Padding**. To create an instance of this class, use the **Cipher.getInstance()** method with "**SAFENET**" as the provider and one of the following strings as the transformation:

> RSA

> RSA/ECB/NoPadding

> RSA/ECB/PKCS1Padding

> RSA/ECB/OAEP

> RSA/ECB/OAEPPadding

Using the "**RSA**" transformation, the Cipher will default to ECB and **PKCS1Padding**. The **NoPadding** option will result in "**RAW**" RSA, where each block will be 0 padded.

The block size of this cipher is dependent on the key size in use. The block size is equal to the number of bytes of the RSA modulus. If the modulus is k bytes long, then the encrypted output size is always k. For the "**NoPadding**" mode, the plaintext input must be equal to or less than k; with the "**PKCS1Padding**" mode, the plaintext input must be equal to or less than k-11 bytes.

This Cipher will only accept a SafeNet ProtectToolkit-J provider-based key during initialization. This key must be generated by the SafeNet ProtectToolkit-J RSA **KeyFactory**, **KeyPairGenerator** or **KeyStore**.

This Cipher does not support initialization with algorithm parameters, and so the **Cipher.getParameters()** method will always return `null`.

RSA Key

The RSA Cipher requires either a SafeNet ProtectToolkit-J RSA public or private Key during initialization. The RSA key may be any length between 512 and 4096 bits (inclusive).

A new SafeNet ProtectToolkit-J RSA key can be generated randomly using the **KeyPairGenerator** as described in section "Public Keys" on page 79, or a provider-independent form as described in section "Key Specifications" on page 84. The RSA key may also be stored in the SafeNet ProtectToolkit-J **KeyStore**, as described in "Key Storage" on page 82 .

The SafeNet ProtectToolkit-J RSA key will return the string "**RSA**" as its algorithm name, the public key type will return "**X.509**" as its encoding (the private key types will return "**RAW**") as its encoding. However, since the key is stored within the hardware, the actual key encoding may not be available (private keys will return `null` from the **getEncoded()** method). If the public key is available, the **getEncoded()** method will return the key as a DER-encoded X.509 **SubjectPublicKeyInfo** block containing the public key as defined in PKCS#1.

The key value can only be extracted from a key if the associated Cryptoki key is not marked as Sensitive. The keys generated in SafeNet ProtectToolkit-J will always be marked as sensitive. It is possible, however, to access any Cryptoki keys stored on the device, and it is possible that the attributes of these keys have been modified.

RSA KeyPairGenerator

The RSA **KeyPairGenerator** is used to generate random RSA key pairs. The generated key pair will consist of two hardware keys, the public key and a private key with the **Cryptoki CKA_SENSITIVE** attribute set. The public exponent for this key generator is fixed to the Fermat-4 value (hex 0x100001).

During initialization, the strength parameter may be any length from 512 to 4096. The default key size is 1024 bits. The random parameter is ignored as the hardware includes a cryptographically-secure random source.

Keys generated using the **KeyPairGenerator** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

RSA KeyFactory

The RSA **KeyFactory** is used to construct SafeNet ProtectToolkit-J keys from their provider-independent form. There are three standard provider-independent forms for RSA keys, one for public keys, and two for private keys. They are:

> **java.security.spec.RSAPublicKeySpec**

> **java.security.spec.RSAPrivateKeySpec**

> **java.security.spec.RSAPrivateCrtKeySpec**

Additionally, there is the **au.com.safenet.crypto.spec.AsciiEncodedKeySpec** class which can be used for keys encoded as hexadecimal strings. For more information on this **KeySpec**, see "Key Specifications" on page 84.

Keys generated using the **KeyFactory** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

To convert one of these supported **KeySpec** classes into a SafeNet ProtectToolkit-J provider key:

```
KeyFactory rsaKeyFact = KeyFactory.getInstance("RSA",
                                            "SAFENET");
PublicKey pubKey = rsaKeyFact.generatePublic(pubKeySpec);
PrivateKey privKey = rsaKeyFact.generatePrivate(privKeySpec);
```

The RSA **KeyFactory** cannot currently convert SafeNet ProtectToolkit-J keys into their provider-independent format, so the **getKeySpec()** method will throw an **InvalidKeySpecException**. The class also cannot perform any key translation via the **translateKey()** method.

RSA Example Code

The following example code will create a random RSA key pair, then create a RSA cipher in ECB mode with **PKCS1Padding**. Next it initializes the cipher for encryption using the public key from a newly-created key pair. Finally, we encrypt the string "**hello world**".

To perform the decryption, we re-initialize the cipher in decrypt mode, with the private key from the key pair.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA",
                                              "SAFENET");
KeyPair rsaPair = keyGen.generateKeyPair();
Cipher rsaCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding",
                                      "SAFENET");
rsaCipher.init(Cipher.ENCRYPT_MODE, rsaPair.getPublic());
byte[] cipherText = rsaCipher.doFinal(
                              "hello world".getBytes());
rsaCipher.init(Cipher.DECRYPT_MODE, rsaPair.getPrivate());
byte[] plainText = rsaCipher.doFinal(cipherText);
```

CHAPTER 5:   Supported Signature Algorithms

The following Signature algorithms are available with the Provider through the **java.security.Signature** interface:

> "MD2withRSA" below
> "MD5withRSA" on the next page
> "SHA1withRSA" on the next page
> "SHA224withRSA" on the next page
> "SHA256withRSA" on the next page
> "SHA384withRSA" on page 68
> "SHA512withRSA" on page 68
> "SHA1withDSA" on page 68
> "PKCS#1RSA" on page 70
> "X.509RSA" on page 70
> "DSARaw" on page 70
> "RIPEMD128withRSA" on page 70
> "RIPEMD160withRSA" on page 70

MD2withRSA

This Signature class implements the algorithm "**MD2withRSA**" as defined in PKCS#1. This algorithm will perform a message digest of the data to be signed, encode that information in a X.509 DigestInfo block, and then RSA encrypt the DER-encoded block.

Initialization requires a SafeNet ProtectToolkit-J RSA key, either a private key for signing or a public key for signature verification. See the RSA Cipher ("RSA" on page 63) for information on RSA keys.

This algorithm is provided for compatibility only; newer applications should use either **MD5withRSA** or **SHA1withRSA**.

The following example will sign the message "**hello world**" with a pre-existing RSA private key, and then verify it with the corresponding public key.

```
KeyPair rsaPair; // pre existing key pair
Signature rsaSig = Signature.getInstance("MD2withRSA", "SAFENET");
rsaSig.initSign(rsaPair.getPrivate());
rsaSig.update("hello world".getBytes());
byte[] sig = rsaSig.sign();
rsaSig.initVerify(rsaPair.getPublic());
rsaSig.update("hello world".getBytes());
if (rsaSig.verify(sig)) {
```

```
   System.out.println("Signature okay");
}
else {
   System.out.println("Signature fails verification");
}
```

MD5withRSA

This Signature class implements the algorithm "**MD5withRSA**", as defined in PKCS#1. This algorithm will perform a message digest of the data to be signed, encode that information in a X.509 DigestInfo block, and then RSA encrypt the DER-encoded block.

Initialization requires a SafeNet ProtectToolkit-J RSA key, either a private key for signing or a public key for signature verification. See the RSA Cipher ("RSA" on page 63) for information on RSA keys.

See "MD2withRSA" on the previous page for a simple example on using this algorithm; modify the algorithm name used to "**MD5withRSA**".

SHA1withRSA

This Signature class implements the algorithm "**RSASSA-PKCS1-v1_5**", as defined in PKCS#1. This algorithm will perform a message digest of the data to be signed, encode that information in a X.509 DigestInfo block and then finally RSA encrypt the DER-encoded block.

Initialization requires a SafeNet ProtectToolkit-J RSA key, either a private key for signing or a public key for signature verification. See the RSA Cipher ("RSA" on page 63) for information on RSA keys.

See "MD2withRSA" on the previous page for a simple example on using this algorithm; modify the algorithm name used to "**SHA1withRSA**".

SHA224withRSA

This signature class is similar to **SHA1withRSA**, except it produces a signature from a digest length of 224 bits.

See "MD2withRSA" on the previous page for a simple example on using this algorithm; modify the algorithm name used to "**SHA224withRSA**".

SHA256withRSA

This signature class is similar to **SHA1withRSA**, except it produces a signature from a digest length of 256 bits.

See "MD2withRSA" on the previous page for a simple example on using this algorithm; simply modify the algorithm name used to "**SHA256withRSA**".

SHA384withRSA

This signature class is similar to **SHA1withRSA**, except it produces a signature from a digest length of 384 bits.

See "MD2withRSA" on page 66 for a simple example on using this algorithm; simply modify the algorithm name used to "**SHA384withRSA**".

SHA512withRSA

This signature class is similar to **SHA1withRSA**, except it produces a signature from a digest length of 512 bits.

See "MD2withRSA" on page 66 for a simple example on using this algorithm; simply modify the algorithm name used to "**SHA512withRSA**".

SHA1withDSA

This Signature class implements the Digital Signature Algorithm (DSA) as defined in *FIPS PUB 186*, which is also compatible with the Sun-provided Signature algorithm of the same name. This algorithm will perform a message digest (using SHA1) of the data to be signed, and then sign that data using DSA. The result of a sign operation using this algorithm will be a DER-encoded block containing a sequence of the two integer values *r* and *s*.

Initialization requires a SafeNet ProtectToolkit-J DSA key, either a private key for signing or a public key for signature verification. The section "DSA Key" below describes how to generate SafeNet ProtectToolkit-J provider DSA keys.

DSA Key

The DSA Signature requires a SafeNet ProtectToolkit-J DSA public or private Key during initialization. The DSA key may be any length between 512 and 4096 bits (inclusive).

A new SafeNet ProtectToolkit-J DSA key pair can be generated randomly using the **KeyPairGenerator**, as described in "Key Generation" on page 78, or, a provider-independent form. The AES key may also be stored in the SafeNet ProtectToolkit-J **KeyStore** as described in "Key Storage" on page 82.

The SafeNet ProtectToolkit-J DSA public and private keys will return the string "**DSA**" as the algorithm name, "**RAW**" as the encoding type and `null` for the encoding.

DSA KeyGenerator

The DSA **KeyPairGenerator** is used to generate random DSA key pairs. The generated key pair will consist of two hardware keys: the public key and a private key with the **Cryptoki CKA_SENSITIVE** attribute set. Each key will also share the same set of DSA parameters.

During initialization, the strength parameter may be either 512 or 4096. The default key size is 1024 bits. The random parameter is ignored as the hardware includes a cryptographically-secure random source. Any provided **AlgorithmParameterSpec** parameters will also be ignored (this precludes generation of keys with non-default parameters). The DSA parameters used for the 512 and 1024 bit keys are as specified in the *Java Cryptography Architecture Specification*.

Keys generated using the **KeyGenerator** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

The following example will generate a new random 1024 bit key pair:

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance(
        "DSA", "SAFENET");
KeyPair dsaPair = keyGen.generateKeyPair();
```

DSA KeyFactory

The DSA **KeyFactory** is used to construct SafeNet ProtectToolkit-J keys from their provider-independent form. There are two standard provider-independent forms for DSA keys: one for public keys and one for private keys. They are **java.security.spec.DSAPublicKeySpec**, and **java.security.spec.DSAPrivateKeySpec**.

Keys generated using the **KeyFactory** are not thread-safe. That is, a SafeNet ProtectToolkit-J Key instance may only be used by a single Cipher instance (as well as a single MAC instance) at any given time. See"Key Generation" on page 78 for information on threading and SafeNet ProtectToolkit-J keys.

To convert one of these supported **KeySpec** classes into a SafeNet ProtectToolkit-J provider key:

```
KeyFactory dsaKeyFact = KeyFactory.getInstance("DSA",
                                       "SAFENET");
PublicKey pubKey = dsaKeyFact.generatePublic(pubKeySpec);
PrivateKey privKey = dsaKeyFact.generatePrivate(privKeySpec);
```
The DSA **KeyFactory** cannot currently convert SafeNet ProtectToolkit-J keys into their provider independent format so the **getKeySpec()** method will throw an **InvalidKeySpecException**. The class also cannot perform any key translation via the **translateKey()** method.

DSA Example Code

The following example code will create a random DSA key pair, then create a DSA Signature. We will then use this instance to sign the message "**hello world**" and verify that signature using the public key.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA",
                                       "SAFENET");
KeyPair rsaPair = keyGen.generateKeyPair();
Signature dsaSig = Signature.getInstance("DSA",
                                       "SAFENET");
dsaSig.initSign(dsaPair.getPrivate());
dsaSig.update("hello world".getBytes());
byte[] sig = dsaSig.sign();
dsaSig.initVerify(dsaPair.getPublic();
dsaSig.update("hello world".getBytes());
if (dsaSig.verify()) {
    System.out.println("Signature okay");
}
else {
    System.out.println("Signature fails verification");
}
```

PKCS#1RSA

This signature algorithm will produce a PKCS#1 encoded block (block type 01) containing the private-key encrypted message. The message length must be *k*-11 bytes long, where *k* is the length of the RSA modulus. The generated signature will be *k* bytes long.

X.509RSA

This signature algorithm will perform "raw" RSA exponentiation on the input message by converting it to an integer (most-significant byte first) and converting the result to a byte string (most-significant byte first). The input message, considered as an integer, must be less than the modulus. Where necessary, the input message is padded by prepending the message with 0-valued bytes.

This algorithm is intended for compatibility with applications that do not follow the PKCS#1 block format.

DSARaw

This signature algorithm will perform "raw" DSA exponentiation on the input message by converting it to an integer (most-significant byte first) and converting the result to a byte string (most-significant byte first). The input message, considered as an integer, must be less than the modulus. Where necessary, the input message is padded by prepending the message with 0-valued bytes.

This algorithm is intended for compatibility with applications that do not follow the PKCS#1 block format.

RIPEMD160withRSA

This Signature class implements the algorithm "**MD5withRSA**", as defined in PKCS#1, with the message digest algorithm **RIPEMD160** in place of MD5. This algorithm will perform a message digest of the data to be signed, encode that information in a X.509 DigestInfo block and then RSA-encrypt the DER-encoded block.

Initialization requires a SafeNet ProtectToolkit-J RSA key, either a private key for signing or a public key for signature verification. See the RSA Cipher ("RSA" on page 63) for information on RSA keys.

See "MD2withRSA" on page 66 for a simple example on using this algorithm; simply modify the algorithm name used to "**RIPEMD128withRSA**".

RIPEMD128withRSA

This Signature class implements the algorithm "**MD5withRSA**", as defined in PKCS#1, with the message digest algorithm **RIPEMD128** in place of MD5. This algorithm will perform a message digest of the data to be signed, encode that information in a X.509 DigestInfo block, and then RSA-encrypt the DER-encoded block.

Initialization requires a SafeNet ProtectToolkit-J RSA key, either a private key for signing or a public key for signature verification. See the RSA Cipher ("RSA" on page 63) for information on RSA keys.

See"MD2withRSA" on page 66 for a simple example on using this algorithm; simply modify the algorithm name used to "**RIPEMD128withRSA**".

CHAPTER 6: Supported MAC Algorithms

The following MAC algorithms are available with the Provider through the **javax.crypto.Mac** interface:

> "DES MAC" below

> "DESede MAC" below

> "DESedeX919 MAC" below

> "IDEA MAC" on the next page

> "CAST128 MAC" on the next page

> "RC2" on the next page

> "HMAC/MD2" on the next page

> "HMAC/MD5" on the next page

> "HMAC/SHA1" on the next page

> "HMAC/SHA224" on page 73

> "HMAC/SHA256" on page 73

> "HMAC/SHA384" on page 73

> "HMAC/SHA512" on page 73

A sample code fragment for generating a MAC code is provided here:

> "Sample MAC Code" on page 73

DES MAC

This algorithm is compatible with *FIPS PUB 113* as well as *ANSI X9.9*.

The MAC may be initialized using any valid DES key (see "DES" on page 44). The result MAC value will be a 4-byte array.

DESede MAC

This algorithm is compatible with *FIPS PUB 113*.

The MAC may be initialized using any valid DESede key (see "DESede" on page 47). The result MAC value will be a 4-byte array.

DESedeX919 MAC

This MAC implements the triple DES MAC algorithm as defined in *X9.19* (or *ISO 9807*).

The MAC may be initialized using any valid DESede key (see "DESede" on page 47). The result MAC value will be a 4-byte array.

IDEA MAC

This algorithm is compatible with *FIPS PUB 113*.

The MAC may be initialized using any valid IDEA key (see "IDEA" on page 52). The result MAC value will be a 4-byte array.

CAST128 MAC

This algorithm is compatible with *FIPS PUB 113*.

The MAC may be initialized using any valid CAST128 key (see "CAST128" on page 54). The result MAC value will be a 4-byte array.

RC2

This algorithm is compatible with *FIPS PUB 113*.

The MAC may be initialized using any valid RC2 key (see "RC2" on page 56). The result MAC value will be a 4-byte array.

HMAC/MD2

This HMAC implements the HMAC algorithm as defined in *RFC 2104* using the message digest function MD2. The result MAC value will be a 16-byte array.

The MAC may be initialized using a **SecretKeySpec** with the algorithm name "HMAC/MD2". It is also possible to initialize this MAC using any of the secret keys generated by one of the **KeyGenerator** classes or **KeyFactory** classes, as detailed in "Supported Ciphers" on page 42.

HMAC/MD5

This HMAC implements the HMAC algorithm as defined in *RFC 2104* using the message digest function MD5. The result MAC value will be a 16-byte array.

The MAC may be initialized using a **SecretKeySpec** with the algorithm name "HMAC/MD5". It is also possible to initialize this MAC using any of the secret keys generated by one of the **KeyGenerator** classes or **KeyFactory** classes, as detailed in "Supported Ciphers" on page 42.

HMAC/SHA1

This HMAC implements the HMAC algorithm as defined in *RFC 2104* using the message digest function SHA1. The result MAC value will be a 20-byte array.

The MAC may be initialized using a **SecretKeySpec** with the algorithm name "HMAC/SHA1". It is also possible to initialize this MAC using any of the secret keys generated by one of the **KeyGenerator** classes or **KeyFactory** classes, as detailed in "Supported Ciphers" on page 42.

HMAC/SHA224

This HMAC implements the HMAC algorithm as defined in *RFC 2104* using the message digest function SHA224. The result MAC value will be a 28-byte array.

The MAC may be initialized using a **SecretKeySpec** with the algorithm name "HMAC/SHA224". It is also possible to initialize this MAC using any of the secret keys generated by one of the **KeyGenerator** classes or **KeyFactory** classes, as detailed in "Supported Ciphers" on page 42.

HMAC/SHA256

This HMAC implements the HMAC algorithm as defined in *RFC 2104* using the message digest function SHA256. The result MAC value will be a 32-byte array.

The MAC may be initialized using a **SecretKeySpec** with the algorithm name "HMAC/SHA256". It is also possible to initialize this MAC using any of the secret keys generated by one of the **KeyGenerator** classes or KeyFactory classes, as detailed in "Supported Ciphers" on page 42.

HMAC/SHA384

This HMAC implements the HMAC algorithm as defined in *RFC 2104* using the message digest function SHA384. The result MAC value will be a 48-byte array.

The MAC may be initialized using a **SecretKeySpec** with the algorithm name "HMAC/SHA384". It is also possible to initialize this MAC using any of the secret keys generated by one of the **KeyGenerator** classes or **KeyFactory** classes, as detailed in "Supported Ciphers" on page 42.

HMAC/SHA512

This HMAC implements the HMAC algorithm as defined in *RFC 2104* using the message digest function SHA512. The result MAC value will be a 64-byte array.

The MAC may be initialized using a **SecretKeySpec** with the algorithm name "HMAC/SHA512". It is also possible to initialize this MAC using any of the secret keys generated by one of the **KeyGenerator** classes or **KeyFactory** classes, as detailed in "Supported Ciphers" on page 42.

Sample MAC Code

This sample code fragment will generate a MAC code (based on a randomly generated DES key) for the bytes in the string "**hello world**".

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES", "SAFENET");
Key desKey = keyGen.generateKey();
```

```
Mac desMac = Mac.getInstance("DES", "SAFENET");
desMac.init(desKey);
byte[] mac = desMac.doFinal("hello world".getBytes());
```

CHAPTER 7:   Supported Message Digest Algorithms

The following standard message digest algorithms are supported by the Provider through the **java.security.MessageDigest** interface:

Message Digest Name	Digest Length (bits)
"MD2" below	128
"MD5" below	128
"SHA-1" on the next page	160
"SHA-224" on the next page	224
"SHA-256" on the next page	256
"SHA-384" on the next page	384
"SHA-512" on the next page	512
"RIPEMD128" on page 77	128
"RIPEMD160" on page 77	160

MD2

This message digest algorithm produces a 128-bit digest. The algorithm is described in *RFC-1319*. This algorithm is provided for compatibility only and is not recommended for other purposes. Instances of this algorithm cannot be cloned.

To create a MD2 message digest for the message "**hello world**":

```
MessageDigest md2 = MessageDigest.getInstance("MD2", "SAFENET");
byte[] digest = md2.digest("hello world".getBytes());
```

MD5

This message digest algorithm produces a 128-bit digest. The algorithm is described in *RFC-1321*. This algorithm is provided for compatibility only and is not recommended for other purposes. Instances of this algorithm cannot be cloned.

To create a MD5 message digest for the message "**hello world**":

```
MessageDigest md5 = MessageDigest.getInstance("MD5", "SAFENET");
byte[] digest = md5.digest("hello world".getBytes());
```

SHA-1

The SHA-1 message digest algorithm produces a 160-bit digest. The algorithm is described in *FIPS PUB 180-1*. Instances of this algorithm cannot be cloned.

To create a SHA-1 message digest for the message "**hello world**":

```
MessageDigest sha1 = MessageDigest.getInstance("SHA-1", "SAFENET");
byte[] digest = sha1.digest("hello world".getBytes());
```

SHA-224

The SHA-224 message digest algorithm produces a 224-bit digest. The algorithm is described in *FIPS PUB 180-1*. Instances of this algorithm cannot be cloned.

To create a SHA-224 message digest for the message "**hello world**":

```
MessageDigest sha256 = MessageDigest.getInstance("SHA-224", "SAFENET");
byte[] digest = sha224.digest("hello world".getBytes());
```

SHA-256

The SHA-256 message digest algorithm produces a 256-bit digest. The algorithm is described in *FIPS PUB 180-1*. Instances of this algorithm cannot be cloned.

To create a SHA-256 message digest for the message "**hello world**":

```
MessageDigest sha256 = MessageDigest.getInstance("SHA-256", "SAFENET");
byte[] digest = sha256.digest("hello world".getBytes());
```

SHA-384

The SHA-384 message digest algorithm produces a 384-bit digest. The algorithm is described in *FIPS PUB 180-1*. Instances of this algorithm cannot be cloned.

To create a SHA-384 message digest for the message "**hello world**":

```
MessageDigest sha384 = MessageDigest.getInstance("SHA-384", "SAFENET");
byte[] digest = sha384.digest("hello world".getBytes());
```

SHA-512

The SHA-512 message digest algorithm produces a 512-bit digest. The algorithm is described in *FIPS PUB 180-1*. Instances of this algorithm cannot be cloned.

To create a SHA-512 message digest for the message "**hello world**":

```
MessageDigest sha512 = MessageDigest.getInstance("SHA-512", "SAFENET");
byte[] digest = sha512.digest("hello world".getBytes());
```

RIPEMD160

The RIPEMD160 message digest algorithm produces a 160-bit digest. Instances of this algorithm cannot be cloned.

To create a RIPEMD160 message digest for the message "**hello world**":

```
MessageDigest rmd160 = MessageDigest.getInstance("RIPEMD160",
 "SAFENET");
byte[] digest = rmd160.digest("hello world".getBytes());
```

RIPEMD128

The RIPEMD128 message digest algorithm produces a 128-bit digest. Instances of this algorithm cannot be cloned.

To create a RIPEMD128 message digest for the message "**hello world**":

```
MessageDigest rmd128 = MessageDigest.getInstance("RIPEMD128",
 "SAFENET");
byte[] digest = rmd128.digest("hello world".getBytes());
```

CHAPTER 8: Key Generation

SafeNet ProtectToolkit-J can generate random keys for each of the cipher algorithms it supports. These keys are Cryptoki session keys; they are not stored permanently on the adapter. Session keys are not thread-safe and so may only be used by a single Cipher instance and a single Signature (or MAC) instance at any time. Thus, it is allowable to use a DES key for encryption in a Cipher instance and a single MAC instance but not two Cipher instances. Keys fetched from the SafeNet ProtectToolkit-J **KeyStore** do not have this restriction.

When generating a random key, the size of the key will be as follows:

Key Name	Default Key Size	Valid Key Sizes
DES	56	56
DESede	196	128,196
AES	128	128,196, 256
IDEA	128	128
CAST128	128	8-128
RC2	64	0-1024
RC4	64	8-2048
RSA	1024	512-4096
DSA	1024	512-3072
DH	1024	512-4096

This section describes the following:

> "Secret Keys" below

> "Public Keys" on the next page

Secret Keys

The secret key Ciphers will simply generate the appropriate number of random bytes for the key (there are no checks for weak keys).

The following example will generate a random double-length DESede key. Generation of a key for a different algorithm is as simple as changing the algorithm name and choosing an appropriate key length.

```
KeyGenerator keyGen = KeyGenerator.getInstance("DESede", "SAFENET");
keyGen.init(128);
SecretKey key = keyGen.generateKey();
```

Public Keys

RSA Keys

The RSA key pair generator will generate keys based on an algorithm determined by key size. If the size is some multiple of 256 bits greater than 1024, the algorithm specified in ANSI X 9.31 will be used. Otherwise, the one specified in PKCS#1 is used. The key pair will be compatible with PKCS#1 RSA, ISO/IEC 9796 RSA and X.509 (raw) RSA standards. ANSI X 9.31 keys have a random 16-bit exponent, while PKCS#1 public exponent is fixed to the Fermat-4 value (hex 0x1001).

The following example will generate a 2048-bit RSA key pair.

```
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA",
                                                "SAFENET");
keyPairGen.initialise(2048);
KeyPair keyPair = keyPairGen.generateKeyPair();
```

DSA Keys

The DSA key pair generator will generate keys based on the algorithm specified in the Digital Signature Standard (*FIPS PUB 186-1*). DSA key generation requires a number of parameters; these are generally fixed in a given application, but they are also usually randomly generated for a particular application. At present, SafeNet ProtectToolkit-J does not include any mechanism to generate these parameters. However, the DSA key pair generator can accept these parameters (via a **java.security.spec.DSAParameterSpec**) or has configured defaults for 512- or 1024-bit keys (these defaults are listed in the JCE specification).

The following example will generate a 1024-bit DSA key pair, using the default DSA parameters.

```
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA",
                                                "SAFENET");
keyPairGen.initialise(1024);
KeyPair keyPair = keyPairGen.generateKeyPair();
```
This example will use the provided DSA parameters, rather than the built-in defaults.

```
BigInteger p, q, g;  // These are the parameter values
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA",
                                                "SAFENET");
DSAParameterSpec keyParamSpec = new DSAParamterSpec(p, q, g);
keyPairGen.initialise(keyParamSpec);
KeyPair keyPair = keyPairGen.generateKeyPair();
```

Diffie-Hellman Keys

The DH **KeyPairGenerator** will generate Diffie-Hellman keys suitable for the Diffie-Hellman key agreement protocol. Diffie-Hellman key generation requires a number of parameters; these are generally fixed in a given application, but they are also usually randomly generated for a particular application. At present, SafeNet ProtectToolkit-J does not include any mechanism to generate these parameters. However, the DH key pair generator can accept these parameters (via a **java.security.spec.DHParameterSpec**) or has configured defaults for 512- or 1024-bit keys (these defaults are listed in the JCE specification).

The following example will generate a 1024-bit DH key pair, using the default DH parameters.

```
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DH",
                                                    "SAFENET");
keyPairGen.initialise(1024);
KeyPair keyPair = keyPairGen.generateKeyPair();
```

This example will use the provided DH parameters, rather than the built-in defaults.

```
BigInteger p, g;  // These are the parameter values
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DH",
                                                    "SAFENET");
DSAParameterSpec keyParamSpec = new DHParamterSpec(p, g);
keyPairGen.initialise(keyParamSpec);
KeyPair keyPair = keyPairGen.generateKeyPair();
```

KeyAgreement Protocols

SafeNet ProtectToolkit-J also includes mechanisms which allow for the creation of keys based on other keys.

Diffie-Hellman KeyAgreement

The DH **KeyAgreement** algorithm can be used to perform a 2-phase key Diffie-Hellman key agreement.

Xor Key Derive

This algorithm may be used to derive a new key from an existing key and a known data pattern. The key value and the data pattern will be combined on the adapter using the XOR function. For example if the initial key has the value 0x12,0x34 and the data pattern has the value 0x89,0xAB, the resultant key will have the value 0x88,0x88.

The actual key values will be combined within the adapter to ensure their values are never compromised. Also, the newly-created key will inherit the attributes of the two keys such that the derived key will be as protected as the two original keys. This mechanism may not be used to change the key type of the base key. Therefore, if the base key is a DES key, the derived key must also be a DES key.

This mechanism can only be used on keys with the **CKA_DERIVE** attribute set to `true`. This will the case for keys generated with any of the SafeNet ProtectToolkit-J mechanisms (such as **KeyGenerator** classes). However, if the key is generated with the Browser application, be sure to check the '**Derive**' checkbox.

Do not create an instance of this class directly, rather use the **KeyAgreement.getInstance()** factory method:

```
KeyAgreement ka = KeyAgreement.getInstance("XorBaseAndKey", "SAFENET");
```

Once created, the instance should be initialized using the base key. Then, to combine with the data pattern, call the **doPhase()** method with a **SecretKeySpec** instance created with the data pattern and true for the **lastPhase** parameter.

Finally to obtain the newly created instance call the **generateSecret()** method with the appropriate key name.

For example:

```
byte[] data = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08};
ka.init(baseKey);
ka.doPhase(new SecretKeySpec(data), true);
Key newKey = ka.generateSecret("DES");
```

NOTE The key material generated must be compatible with the key type requested in the **generateSecret()** method call. Specifically, the length of the new key will be the minimum of the lengths of the two components.

CHAPTER 9:   Key Management

This section provides information on the following:

> "Key Storage" below
> "Key Wrapping" on the next page
> "Key Specifications" on page 84

Key Storage

The encryption adapter has the facility to store public, private, and secret keys. These keys will be stored in the non-volatile storage on the card. As well as key storage, it is also possible to store X.509 Certificates (which contain a public key). SafeNet ProtectToolkit-J provides access to this storage mechanism via the JCE KeyStore API. The JCE name for this KeyStore is CRYPTOKI.

The JCE KeyStore API allows storage of a Key and an associated alias. This alias is simply a unique string which may be used to access the key. To store a key in the key store, use the **setKeyEntry()**. To retrieve a key, use the **getKey()**. Keys may be removed from the KeyStore using the **deleteEntry()** method.

Currently, only two types of keys may be stored in the SafeNet ProtectToolkit-J KeyStore: either SafeNet ProtectToolkit-J keys or **javax.crypto.spec.SecretKeySpec** keys. Other key types must be converted to their SafeNet ProtectToolkit-J equivalents before storage.

Currently, the Certificate support is based on Sun's Certificate implementation which is only available on the Sun Java2 JVM.

Per Key password protection is not supported, so a null password may be supplied to the methods used to store and retrieve keys from the KeyStore. The password provided to the **load()** method will be used to log in to the token, and so to access private objects on the token it is necessary to provide the PIN. If a PIN is not supplied, all objects will be stored as public objects. When a PIN is supplied, only PublicKey and Certificate objects will be stored as public objects; all others will be private. In either case, the InputStream passed to the **store()** and **load()** methods will not change the contents of the key store.

Keys stored in the KeyStore are the only thread-safe SafeNet ProtectToolkit-J keys. A key instance obtained from the **KeyStore.getKeyEntry()** method will return a key that may be used in multiple Cipher, MAC, and Signature instances.

The following example will create a new random DES key, and then store that key in the KeyStore. Note that even though we first create the key and then store it, the actual key value will not leave the hardware and therefore remains secure.

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES", "SAFENET");
Key key = keyGen.generateKey();
KeyStore keyStore = KeyStore.getInstance("CRYPTOKI", "SAFENET");
keyStore.load(null, null);
keyStore.setKeyEntry("des key", key, null, null);
```
The following example can be used to access the previously stored key:

```
KeyStore keyStore = KeyStore.getInstance("CRYPTOKI", "SAFENET");
keyStore.load(null, null);
Key key = keyStore.getKey("des key", null);
```

Key Wrapping

The CRYPTOKI KeyStore also provides a key wrapping mechanism. Key wrapping is a technique where one key value is encrypted using another key. With SafeNet ProtectToolkit-J, since the key values are stored securely on the hardware, we can use this technique to encrypt the key on the hardware and then extract the encrypted key.

For example, using this mechanism, a session key may be generated on the hardware and then exported from the hardware in an encrypted (wrapped) form. The key will generally be encrypted using a Public/Private key encryption cipher and can then be safely exported from the HSM. It is also possible to use secret keys for key wrapping. In this case, however, the same secret key must exist on both the source (performing the wrapping function) and the destination adapters.

The WrappingKeyStore API is an extension to the standard JCE that is used to provide access to key wrapping services. This class is identical to the standard KeyStore API, except that it provides **wrapKey()** and **unwrapKey()**methods. The wrapping key store can be instantiated using the following code:

```
import au.com. safenet.crypto.WrappingKeyStore;

...

WrappingKeyStore wks = WrappingKeyStore.getInstance("CRYPTOKI",
                                                    "SAFENET");
...
```

The **wrapKey()** method has the following signature:

```
public byte[] wrapKey(Key wrapKey, String transformation, Key key)
throws GeneralSecurityException
```

The wrapKey parameter specifies the Key used to encrypt the key parameter. The transformation parameter specifies the encryption transformation that is to be used to encrypt the key. With the CRYPTOKI KeyStore, you can transform the following:

> RSA/ECB/PKCS1Padding

> RSA/ECB/NoPadding

> DES/ECB/NoPadding

> DES/ECB/PKCS5Padding

> DESede/ECB/NoPadding

> DESede/ECB/PKCS5Padding

> IDEA/ECB/NoPadding

> IDEA/ECB/PKCS5Padding

> CAST128/ECB/NoPadding

> CAST128/ECB/PKCS5Padding

> RC2/ECB/NoPadding

> RC2/ECB/PKCS5Padding

> RC4

A GeneralSecurityException will be thrown if the transformation parameter is invalid.

The value returned is a byte array containing the encrypted key. This value may be passed to the **unwrapKey ()**method to extract the original key. The **unwrapKey()** method has the following signature:

```
public Key unwrapKey(Key unwrapKey, String transformation,
                     byte[] wrappedKey, String keyAlgorithm)
throws GeneralSecurityException
```

This method will "unwrap" or decrypt the encrypted key using the provided decryption key and transformation. The Key returned will be of the type specified by the keyAlgorithm parameter. This parameter must match the actual key type that was originally wrapped.

The unwrapKey parameter should be either the same secret key as was used to wrap the key, or the private key corresponding to the public key used to wrap the key. The transformation parameter specifies the decryption transformation used to decrypt the key. This value should be the same as that used to wrap the key. The wrappedKey parameter should contain the encrypted key. The keyAlgorithm should specify the algorithm that the decrypted key is for.

A GeneralSecurityException will be thrown if the transformation parameter is invalid.

The following example will create a new random RC4 key, wrap that key with an RSA public key, and unwrap it with the associated RSA private key.

```
KeyGenerator keyGen = KeyGenerator.getInstance("RC4", "SAFENET");
Key rc4Key = keyGen.generateKey();
WrappingKeyStore wks = WrappingKeyStore.getInstance("CRYPTOKI");
wks.load(null, null);  // initialise the KeyStore
Key publicKey = wks.getKey("RSA_pub", null);
byte[] encKey = Wks.wrapKey(publicKey,"RSA/ECB/PKCS1Padding",rc4Key);
// give the encrypted key to the recipient, and unwrap it
Key privateKey = wks.getKey("RSA_priv", null);
Key recoveredKey = wks.unwrapKey(privateKey, "RSA/ECB/PKCS1Padding',
                                                encKey);
```

Key Specifications

As well as supporting the relevant JCA/JCE defined KeySpec classes, SafeNet ProtectToolkit-J includes a number of custom provider-independent key classes for use with its KeyFactory classes. These classes all live in the **au.com.safenet.crypto.spec** package:

AsciiEncodedKeySpec

Used to encode RSA, DSA or Diffie-Hellman public and private keys as ASCII strings. These strings contain the key's integer components as hexadecimal strings separated by a full stop. For example, an RSA private key:

```
public_exponent.modulus.private_exponent.p.q
```

A public key will contain only the first two elements and a private key will contain all five. The RSA KeyFactory can convert from this KeySpec into the provider-based key.

For DSA keys the format is:

y.p.q.g (private keys) x.p.q.g (public keys)

For Diffie-Hellman keys, the format is:

y.p.g (private keys) x.p.g (public keys)

CASTKeySpec

Used to encode keys for the CAST algorithm. This class takes a byte array, which it will use directly as the CAST key. The array must be less than or equal to 16 bytes, the maximum key size for a CAST key.

IDEAKeySpec

Used to encode keys for the IDEA algorithm. This class takes a byte array and uses the first 16 bytes of the array as the IDEA key.

RC2KeySpec

Used to encode keys for the RC2 algorithm. This class takes a byte array, which it will use directly as the RC2 key. The array must be less than or equal to 128 bytes, the maximum key size for a RC2 key.

RC4KeySpec

Used to encode keys for the RC4 algorithm. This class takes a byte array, which it will use directly as the RC4 key. The array must be less than or equal to 256 bytes, the maximum key size for a RC4 key.

AESKeySpec

Used to encode keys for the AES algorithm. This class takes a byte array, which it will use directly as the AES key. The array must be 16, 24 or 32 bytes.

CHAPTER 10: Best Practice Guidelines

The purpose of this section is to outline some of the best practices application developers can use when developing their SafeNet ProtectToolkit-J based applications.

The following guidelines do not attempt to replace the vast body of literature regarding building secure systems or implementing cryptography for security. Rather it focuses on some of the specific aspects of the SafeNet ProtectToolkit-J product that are particularly relevant to building applications in a timely and reliable way.

SafeNet ProtectToolkit-J Provider

The SafeNet ProtectToolkit-J JCA/JCE Provider provides access to the many cryptographic features of the SafeNet ProtectServer range of hardware.

As the provider is hardware-based, there are a number of differences between it and other software-based implementations. Mostly, these stem from the different methods used to protect the key store, where hardware can effectively provide some level of physical protection.

Key Protection

Usage

Each key has an associated set of usage flags that indicate which cryptographic operations may be performed with the key. For example, specific flags may be set to enable encryption or signature generation. Keys in the SafeNet ProtectToolkit-J provider will adhere to these rules.

Value

Normally, keys protected by the hardware will not allow their values to be revealed outside the adapter. Thus, the **Key.getEncoded()** interface will generally return a `null` value.

General SafeNet ProtectToolkit-J Usage Guidelines

> Create persistent keys with the Key Management Utility (KMU) and specify their key usage attributes appropriately.

- secret and private keys should always be sensitive

- each key should be usable for only one purpose

- use the KMU for key backups with the exportable attribute

> Persistent key instances from the SafeNet ProtectToolkit-J KeyStore implementation are shareable. This means a key lookup only needs to be performed once, rather than every time a key is required.

> Initialize the token correctly. Different applications should use different tokens.

> Install the SafeNet ProtectToolkit-J provider as the highest priority, or use **Security.insertProvider (SAFENETProvider()** early on in your application. This will ensure that the SAFENET hardware SecureRandom will become the system default, providing improved quality random data and avoiding the startup performance penalty of the Sun implementation.

> Fully specify Cipher transformations. For example, use "DES/ECB/NoPadding" instead of "DES".

APPENDIX A: JCA/JCE API Tutorial

This appendix will introduce the reader to the Java API known as the Java Cryptography Extension (JCE) through development of a simple application.

It is important to note that this tutorial does not provide complete coverage of this API. The API specification documentation should serve as the detailed reference. It can be found here: http://docs.oracle.com/

During this tutorial we will develop a JCE-based application that allows for simple file encryption. This application will allow the user to encrypt and decrypt files.

The files are encrypted using a combination of public-key and secret-key cryptography. The encrypted files also include a Message Authentication Code (MAC) to ensure the integrity of their contents. Where possible, the standard API mechanisms will be used to achieve the desired functionality.

The code fragments included in this document are used to highlight the important sections of the application. The full source code for the application may be found in the Java source file **FileCrypt.java**.

This document contains the following chapters:

> "Public Key Cryptography" below

> "FileCrypt Application" below

- "File Encryption" on the next page
- "File Decryption" on page 93
- "Accessing Public Keys" on page 98
- "Main()" on page 98

Public Key Cryptography

The sample application will encrypt a document using a secret-key cipher algorithm, for example DES or RC4, and a randomly generated key. This algorithm is known as the bulk cipher, as it is used to perform the bulk of the encryption. The randomly generated key will be encrypted using a public-key cipher algorithm.

By combining public-key and secret-key encryption in this manner, we retain the advantages of public-key cryptography (we don't have to share a secret key) and the performance advantage of a secret-key cipher.

It is assumed that two public key pairs have been generated for this application: the first for the document sender and the second for the recipient.

FileCrypt Application

The **FileCrypt** application enables files to be encrypted for a given recipient and then decrypted by that recipient. Since the encrypted file contains a MAC, the recipient of a document will also be able to verify that the encrypted file was not tampered with.

These encrypted files will be stored in this custom format:

Field	Length (bytes)
KeyLength	4
KeyBytes	As specified by KeyLength
AlgParamsLength	4
AlgParams	As specified by AlgParamsLength
MacLength	4
Mac	As specified by MacLength
Encrypted Data	Remainder of file

This section contains information on the following functions:

> "File Encryption" below

> "File Decryption" on page 93

> "Accessing Public Keys" on page 98

> "Main()" on page 98

File Encryption

In order to encrypt a file, we need to know the public key of its recipient - the party who can decrypt the file. These arguments are passed to the **encryptFile()** method.

The **encryptFile()** method will:

1. generate a random session key,

2. encrypt the session key with the recipient's public key,

3. initialize the bulk cipher with the session key,

4. encode the bulk cipher's algorithm parameters,

5. initialize the MAC algorithm,

6. process the input file, and

7. create the output from the various components.

Step 1 - Generate a Random Session Key

To achieve acceptable performance during file encryption and decryption, we need to use a symmetric-key cipher. This symmetric key, which we will call the session key, will be encrypted (using the recipient's public key) and then stored with the encrypted file. Rather than simply using the same key for each file, we need to generate a random key for each encryption.

The **KeyGenerator** mechanism is used to create random **SecretKey** key objects. A provider-based instance is created using the **KeyGenerator.getInstance()** method.

This instance can then be initialized using one of the **KeyGenerator.init()** methods. In the simplest case, no initialization is required, in which case the provider's default initialization is used. Alternatively, initialization can request a key of the given key size, or other key parameters by using a **java.security.AlgorithmParameterSpec** class.

The following method will create a new random **SecretKey** for the given algorithm and provider using the default initialization:

```
SecretKey generateSecretKey(String algorithm, String
                            provider)
{
   KeyGenerator keyGen = KeyGenerator.getInstance(
      algorithm, provider);

   return keyGen.generateKey();
}
```

Step 2 - Encrypt the Session Key

Once we have generated the session key, we need to encrypt it using the recipient's public key. In this way we can safely transmit the session key such that only the recipient can recover the actual key. The Thales **SAFENET** provider includes a special interface to its **KeyStore** to provide session key encryption.

The **au.com.safenet.crypto.WrappingKeyStore** class extends the standard **KeyStore** mechanism to provide "key wrapping" which enables a session key to be generated in the hardware, then encrypted on the hardware and exported in an encrypted form. This means that the session key is never visible outside the hardware.

The **WrappingKeyStore.wrapKey()** method accepts three arguments: two keys and a transformation string. The first Key is the RSA **PublicKey** used to perform the encryption, the second Key is the DES key we wish to encrypt. The final parameter, the transformation string, describes the encryption method that should be used to encrypt the key. Currently, this string may be **RSA/ECB/PKCS1Padding** or **RSA/ECB/NoPadding**.

```
static final String PROVIDER = "SAFENET";
static final String WRAP_KEYSTORE = "CRYPTOKI";
static final String WRAP_TRANSFORM =
                   "RSA/ECB/PKCS1Padding";

byte[] encryptKey(PublicKey wrapKey, SecretKey key)
{
   WrappingKeyStore keyStore;
   keyStore = WrappingKeyStore.getInstance(WRAP_KEYSTORE,
                                    PROVIDER);
   keyStore.load(null, null);
   return keyStore.wrapKey(wrapKey, WRAP_TRANSFORM, key);
}
```

Step 3 - Create and Initialize the Bulk Cipher

This application will simply use the default **AlgorithmParameters** for the bulk encryption algorithm. Therefore, the initialization of our **Cipher** is quite simple:

```
static final String PROVIDER = "SAFENET";
static final String BULK_ALGORITHM = "DES";
```

```
Cipher bulkCipher = Cipher.getInstance(BULK_ALGORITHM,
                                       PROVIDER);
bulkCipher.init(Cipher.ENCRYPT_MODE, secretKey);
```

Step 4 - Encode Algorithm Parameters

The only algorithm parameter supported by the Thales **SAFENET** provider is an initialization vector. An initialization vector is used in a block cipher when it is operating in a feedback mode: DES in CBC mode for example. During encryption, the initialization vector is used to prime the cipher. However, unlike the key, its value is not secret.

The cipher used to decrypt the data stream must be initialized with the same initialization vector for the decryption to succeed.

The following method will return the algorithm parameters encoded into a byte array. For now, we just return the IV directly as this is the only supported algorithm parameter.

```
byte[] encodeParameters(Cipher cipher)
{
   byte[] iv = cipher.getIV();
   return iv;
}
```

Step 5 - Initialize the MAC Algorithm

In this example we will use a MAC algorithm instead of a signature algorithm. The significant difference here is that the MAC will only tell us if the encrypted document has been tampered with, it will not authenticate the sender.

```
static final String PROVIDER = "SAFENET";
static final String MAC_ALGORITHM = "DESMac";

Mac mac = Mac.getInstance(MAC_ALGORITHM, PROVIDER);
mac.init(secretKey);
```

Step 6 - Process the Input File

We are now ready to process the input file to generate the encrypted output and the MAC. The following method will accept the initialized **Cipher**, **Mac** and input/output streams. The data on the **InputStream** will be read in blocks (of some arbitrary size), then processed by the **Mac** instance and then encrypted with the **Cipher** instance.

The encrypted data will then be written to the **OutputStream**. This method will return the MAC as a byte array.

```
static final int READ_BUFFER = 50;

byte[] encrypt(Cipher cipher, Mac mac, InputStream in,
              OutputStream out)
{
   byte[] block = new byte[READ_BUFFER];
   int len;
   while ((len = in.read(block)) != -1)
   {
      /*
       * update our MAC value
       */
      mac.update(block, 0, len);
```

```
    /*
     * encrypt the data
     */
    byte[] enc = cipher.update(block, 0, len);
    if (enc != null)
    {
        /*
         * output the encrypted data
         */
        out.write(enc);
    }
}

/*
 * output the final block if required
 */
byte[] finalBlock = cipher.doFinal();
if (finalBlock != null)
{
    out.write(finalBlock);
}

return mac.doFinal();
}
```

Step 7 - Create the Encrypted Output

Now that we have written the various building blocks, we can construct the final **encryptFile()** method:

```
static final String PROVIDER = "SAFENET";
static final String BULK_ALGORITHM = "DES";
static final String BULK_TRANSFORM =
                   "DES/CBC/PKCS5Padding";
static final String MAC_ALGORITHM = "DESMac";

void encryptFile(InputStream in, OutputStream out,
                 PublicKey publicKey)
{
    /*
     * Create a random SecretKey and encrypt it using
     * the recipient's PublicKey
     */
    SecretKey secretKey = generateSecretKey(BULK_ALGORITHM,
                                            PROVIDER);
    byte[] wrappedKey = encryptKey(publicKey, secretKey);

    /*
     * Create and initialise the Cipher used to encrypt the
       document
     */
    Cipher bulkCipher =
            Cipher.getInstance(BULK_TRANSFORM,PROVIDER);
    bulkCipher.init(Cipher.ENCRYPT_MODE, secretKey);

    /*
     * Encode the algorithm parameters for the Cipher
     */
    byte[] algParams = encodeParameters(bulkCipher);
```

```
   /*
    * Create the Mac instance and initialise it with our
    * session key
    */
   Mac mac = Mac.getInstance(MAC_ALGORITHM, PROVIDER);
   mac.init(secretKey);

   /*
    * Encrypt the document to an internal buffer and
    * calculate the MAC value of the plain text
    */
   ByteArrayOutputStream bOut =
                            new ByteArrayOutputStream();
   byte[] macValue = encrypt(bulkCipher, mac, in, bOut);

   /*
    * Encode the output file
    */
   DataOutputStream dOut = new DataOutputStream(out);

   /*
    * Write out the key
    */
   dOut.writeInt(wrappedKey.length);
   dOut.write(wrappedKey);

   /*
    * Write out the parameters, note these may be null
    */
   if (algParams != null)
   {
      dOut.writeInt(algParams.length);
      dOut.write(algParams);
   }
   else
   {
      dOut.writeInt(0);
   }

   /*
    * Write out the MAC
    */
   dOut.writeInt(macValue.length);
   dOut.write(macValue);

   /*
    * And finally the encrypted document
    */
   bOut.writeTo(dOut);
}
```

File Decryption

To decrypt an encrypted file we simply need to reverse the encryption process. However, rather than using the recipient's public key, we need to use the private key in order to recover the session key.

The **decryptFile()** method will:

1. decode the input from the various components and decipher the session key with the recipient's private key,

2. initialize the bulk cipher with the session key and algorithm parameters,

3. initialize the MAC algorithm,

4. process the encrypted input,

5. verify the calculated MAC with the MAC from the document, and

6. write out the decrypted result.

Step 1 - Decrypt the session key

```
static final String PROVIDER = "SAFENET";
static final String WRAP_KEYSTORE = "CRYPTOKI";
static final String WRAP_TRANSFORM = "RSA/ECB/PKCS1Padding";
static final String BULK_ALGORITHM = "DES";

Key decryptKey(PrivateKey wrapKey, byte[] wrappedKey)
{
   WrappingKeyStore keyStore;
   keyStore = WrappingKeyStore.getInstance(WRAP_KEYSTORE,
                                           PROVIDER);

   return keyStore.unwrapKey(wrapKey, WRAP_TRANSFORM,
                             wrappedKey, BULK_ALGORITHM);
}
```

Step 2 - Initialize the Bulk Cipher

Next, we need to create and initialize the **Cipher** instance we will use to decrypt the document. It is important here to ensure that our **Cipher** instance that will be used to perform the decryption is initialized with the same parameters generated by the encryption **Cipher**. In the case of the Thales **SAFENET** provider, the only parameter type is the **IvParameterSpec**, so we convert our serialized parameters directly.

```
static final String PROVIDER = "SAFENET";
static final String BULK_ALGORITHM = "DES";

Cipher bulkCipher = Cipher.getInstance(BULK_TRANSFORM,
                                       PROVIDER);
if (algParams != null)
{
   AlgorithmParameterSpec params;
   params = new IvParameterSpec(algParams);

   bulkCipher.init(Cipher.DECRYPT_MODE, secretKey,
                   params);
}
else
{
   bulkCipher.init(Cipher.DECRYPT_MODE, secretKey);
}
```

Step 3 - Initialize the MAC Algorithm

Initialization of the MAC during decryption is identical to that during encryption:

```
static final String PROVIDER = "SAFENET";
static final String MAC_ALGORITHM = "DESMac";

    Mac mac = Mac.getInstance(MAC_ALGORITHM, PROVIDER);
    mac.init(secretKey);
```

Step 4 - Process the encrypted input

Next we need to recover the plaintext from the ciphertext and calculate a new MAC. This process is nearly identical to the **encrypt()** method, however, since the MAC is calculated on the plaintext, we update the Mac with the output from the **Cipher**.

```
static final int READ_BUFFER = 50;

byte[] decrypt(Cipher cipher, Mac mac, InputStream in, OutputStream out)
{
    /*
     * read the input in chunks and process each chunk
     */
    byte[] block = new byte[READ_BUFFER];
    int len;
    while ((len = in.read(block)) != -1)
    {
        /*
         * decipher the data
         */
        byte[] plain = cipher.update(block, 0, len);
        if (plain != null)
        {
            /*
             * update our MAC value
             */
            mac.update(plain);

            /*
             * output the deciphered data
             */
            out.write(plain);
        }
    }

    /*
     * output the final block if required
     */
    byte[] finalBlock = cipher.doFinal();
    if (finalBlock != null)
    {
        /*
         * update our MAC value
         */
        mac.update(finalBlock);

        /*
         * output the deciphered data
         */
        out.write(finalBlock);
    }
```

```
    return mac.doFinal();
}
```

Step 5 - Verify the MAC

To verify the MAC, we simply compare the MAC bytes we previously extracted with the value just calculated.

```
if (!Arrays.equals(fileMac, calculatedMac))

{
    throw new GeneralSecurityException("File has been
    tampered with.");
}
```

Step 6 - Write out the decrypted result

Now that we have verified that the file is not corrupted we can output the contents to the destination.

```
static final String PROVIDER = "SAFENET";
static final String BULK_ALGORITHM = "DES";
static final String BULK_TRANSFORM = "DES/CBC/PKCS5Padding";
static final String MAC_ALGORITHM = "DESMac";

void decryptFile(InputStream in, OutputStream out,PrivateKey privateKey)
{
    /*
     * Decode the input file
     */
    DataInputStream dIn = new DataInputStream(in);

    /*
     * recover the encrypted Key data
     */
    int keyLen = dIn.readInt();
    byte[] keyBytes = new byte[keyLen];
    dIn.readFully(keyBytes);

    /*
     * recover the algorithm parameters
     */
    int algLen = dIn.readInt();
    byte[] algBytes = null;
    if (algLen > 0)
    {
        algBytes = new byte[algLen];
        dIn.readFully(algBytes);
    }

    /*
     * recover the stored MAC value
     */
    int macLen = dIn.readInt();
    byte[] fileMac = new byte[macLen];
    dIn.readFully(fileMac);

    /*
     * recreate the session key
     */
```

```
Key secretKey = decryptKey(privateKey, keyBytes);

/*
 * Create our Cipher and initialise it with our key
 * and algorithm parameters.
 */
Cipher bulkCipher =
            Cipher.getInstance(BULK_TRANSFORM,PROVIDER);
if (algBytes != null)
{
    AlgorithmParameterSpec params;
    params = new IvParameterSpec(algBytes);

    bulkCipher.init(Cipher.DECRYPT_MODE, secretKey,
                    params);
}
else
{
    bulkCipher.init(Cipher.DECRYPT_MODE, secretKey);
}

/*
 * Initialise the Mac we use to verify the file
   integrity
 */
Mac mac = Mac.getInstance(MAC_ALGORITHM, PROVIDER);
mac.init(secretKey);

/*
 * Decrypt the file to a temporary buffer
 */
ByteArrayOutputStream bOut =
                        new ByteArrayOutputStream();
byte[] calculatedMac = decrypt(bulkCipher, mac, in,
                               bOut);

/*
 * verify the stored MAC value with the calculated
   value
 */
if (!Arrays.equals(fileMac, calculatedMac))
{
    throw new GeneralSecurityException(
        "File has been tampered with.");
}
else
{
    /*
     * save the decrypted output to the outputstream
     */
    bOut.writeTo(out);
}
}
```

Accessing Public Keys

A Java **java.security.KeyStore** implementation is used to store the public keys for this application. The Thales **SAFENET** provider implementation of the **KeyStore** is known as **CRYPTOKI**, and enables access to the keys stored on the hardware. At present, this **KeyStore** only supports storage of Key objects and does not provide any support for the storage of Certificate objects. Additionally, this **KeyStore** will ignore the password parameter supplied to the **getKey()** method.

Creating the KeyStore

Creating a **KeyStore** instance and populating it is generally a two step process. First, we create the instance and then use the **KeyStore.load()** method to initialize it with the key data. The **load()** method accepts an **InputStream** instance which allows for keys to be stored on an arbitrary data source. The **CRYPTOKI KeyStore**, however, accesses key storage on the hardware directly and so ignores the **load()** method completely.

```
static final String PROVIDER = "SAFENET";
static final String KS_NAME = "CRYPTOKI";

KeyStore loadKeyStore()
{
   KeyStore ks = KeyStore.getInstance(KS_NAME, PROVIDER);
   ks.load(null, null);

   return ks;
}
```

Retrieving the Public Key

Our application needs to determine the recipient's public key in order to encrypt the file. The standard mechanism for accessing public keys is to extract the Certificate for the recipient by using the **KeyStore.getCertificate()** method and then use the **Certificate.getPublicKey** method to recover the key. However with the **CRYPTOKI KeyStore** we will simply use the **KeyStore.getKey()** method.

```
PublicKey publicKey = (PublicKey)ks.getKey(recipientAlias,
                   null);
```

Retrieving the Private Key

To decrypt the file we need to look up the private key. To access private keys stored in a **KeyStore** use the **KeyStore.getKey()** method.

```
PrivateKey privateKey = (PrivateKey)ks.getKey(myAlias,
                     null);
```

Main()

Now that we have all the required building blocks, the last remaining step is to put it all together. We need to process command line arguments and call the appropriate methods. We also need to add exception handling.

The following **main()** method is responsible for determining if we are encrypting or decrypting the file and the names of the keys to use:

```
public static void main(String[] args)
{
   boolean encrypt = false;
   boolean decrypt = false;

   String keyName = null;

   /*
    * examine all the command line arguments
    */
   for (int i = 0; i < args.length; i++)
   {
      if (args[i].equals("-encrypt"))
      {
         encrypt = true;
      }
      else if (args[i].equals("-decrypt"))
      {
         decrypt = true;
      }
      else if (args[i].equals("-key"))
      {
         keyName = args[++i];
      }
   }

   /*
    * validate the arguments
    */
   if (encrypt == decrypt)
   {
      if (encrypt)
      {
         System.err.println("Cannot encrypt and decrypt
         file!");
      }
      else
      {
         System.err.println("Must specify -encrypt or -
         decrypt.");
      }
      System.exit(1);
   }

   if (keyName == null)
   {
      System.err.println("Missing key name.");
      System.exit(1);
   }

   FileCrypt fileCrypt = new FileCrypt();
   KeyStore ks = fileCrypt.loadKeyStore();

   if (encrypt)
   {
      PublicKey publicKey = (PublicKey)ks.getKey(keyName,
                                                  null);

      fileCrypt.encryptFile(System.in, System.out, publicKey);
```

```
    }
    else
    {
        PrivateKey privateKey = (PrivateKey)ks.getKey(keyName,
        null);
        fileCrypt.decryptFile(System.in, System.out, privateKey);
    }
}
```

APPENDIX B: Random Number Generation

The Safenet provider (named "**safenet**") implements a **java.security.SecureRandom** class for generating random data. This implementation is known as "CRYPTOKI". Besides using a hardware-based entropy generator, one of the major benefits of this implementation is that it does not suffer from the slow initialization problem that the Sun-provided (and most other) software implementations do.

This interface is only available under Java2.

This implementation allows access to the encryption adapter random source for both seeding and random number generation. The SafeNet ProtectServer PCIe HSM uses hardware-based random number generation.

Serialization of an instance of this class will not save the state of the random number generator as it is contained within the hardware.

APPENDIX C: References

This section contains a list of resources used as references in this guide:

> *FIPS PUB 42-2*: Data Encryption Standard
> *FIPS PUB 81*: DES Modes of Operation
> *FIPS PUB 113*: Computer Data Authentication
> *FIPS PUB 180-1*: Secure Hash Standard
> *FIPS PUB 186-1*: Digital Signature Standard (DSS)
> *PKCS#1*: RSA Encryption Standard
> *PKCS#5*: Block Cipher Padding
> *PKCS#11*: Cryptographic Token Interface Standard
> *RFC-1319*: The MD2 Message-Digest Algorithm
> *RFC-2104*: HMAC - Keyed-Hashing for Message Authentication
> *RFC-2144*: The Cast-128 Encryption Algorithm
> *RFC-2268*: A Description of the RC2(r) Encryption Algorithm
> *RFC-3281*: An Internet Attribute Certificate Profile for Authorization

Glossary

A

Adapter

The printed circuit board responsible for cryptographic processing in a HSM

AES

Advanced Encryption Standard

API

Application Programming Interface

ASO

Administration Security Officer

Asymmetric Cipher

An encryption algorithm that uses different keys for encryption and decryption. These ciphers are usually also known as public-key ciphers as one of the keys is generally public and the other is private. RSA and ElGamal are two asymmetric algorithms

B

Block Cipher

A cipher that processes input in a fixed block size greater than 8 bits. A common block size is 64 bits

Bus

One of the sets of conductors (wires, PCB tracks or connections) in an IC

C

CA

Certification Authority

CAST

Encryption algorithm developed by Carlisle Adams and Stafford Tavares

Certificate

A binding of an identity (individual, group, etc.) to a public key which is generally signed by another identity. A certificate chain is a list of certificates that indicates a chain of trust, i.e. the second certificate has signed the first, the

third has signed the second and so on

CMOS

Complementary Metal-Oxide Semiconductor. A common data storage component

Cprov

ProtectToolkit C - SafeNet's PKCS #11 Cryptoki Provider

Cryptoki

Cryptographic Token Interface Standard. (aka PKCS#11)

CSA

Cryptographic Services Adapter

CSPs

Microsoft Cryptographic Service Providers

D

Decryption

The process of recovering the plaintext from the ciphertext

DES

Cryptographic algorithm named as the Data Encryption Standard

Digital Signature

A mechanism that allows a recipient or third party to verify the originator of a document and to ensure that the document has not be altered in transit

DLL

Dynamically Linked Library. A library which is linked to application programs when they are loaded or run rather than as the final phase of compilation

DSA

Digital Signature Algorithm

E

Encryption

The process of converting the plaintext data into the ciphertext so that the content of the data is no longer obvious. Some algorithms perform this function in such a way that there is no known mechanism, other than decryption with the appropriate key, to recover the plaintext. With other algorithms there are known flaws which reduce the difficulty in recovering the plaintext

F

FIPS

Federal Information Protection Standards

FM

Functionality Module. A segment of custom program code operating inside the CSA800 HSM to provide additional or changed functionality of the hardware

FMSW

Functionality Module Dispatch Switcher

H

HA

High Availability

HIFACE

Host Interface. It is used to communicate with the host system

HSM

Hardware Security Module

I

IDEA

International Data Encryption Algorithm

IIS

Microsoft Internet Information Services

IP

Internet Protocol

J

JCA

Java Cryptography Architecture

JCE

Java Cryptography Extension

K

Keyset

A keyset is the definition given to an allocated memory space on the HSM. It contains the key information for a specific user

KWRAP

Key Wrapping Key

M

MAC

Message authentication code. A mechanism that allows a recipient of a message to determine if a message has been tampered with. Broadly there are two types of MAC algorithms, one is based on symmetric encryption algorithms and the second is based on Message Digest algorithms. This second class of MAC algorithms are known as HMAC algorithms. A DES based MAC is defined in FIPS PUB 113, see http://www.itl.nist.gov/div897/pubs/fip113.htm. For information on HMAC algorithms see RFC-2104 at http://www.ietf.org/rfc/rfc2104.txt

Message Digest

A condensed representation of a data stream. A message digest will convert an arbitrary data stream into a fixed size output. This output will always be the same for the same input stream however the input cannot be reconstructed from the digest

MSCAPI

Microsoft Cryptographic API

MSDN

Microsoft Developer Network

P

Padding

A mechanism for extending the input data so that it is of the required size for a block cipher. The PKCS documents contain details on the most common padding mechanisms of PKCS#1 and PKCS#5

PCI

Peripheral Component Interconnect

PEM

Privacy Enhanced Mail

PIN

Personal Identification Number

PKCS

Public Key Cryptographic Standard. A set of standards developed by RSA Laboratories for Public Key Cryptographic processing

PKCS #11

Cryptographic Token Interface Standard developed by RSA Laboratories

PKI

Public Key Infrastructure

ProtectServer

SafeNet HSM

ProtectToolkit C

SafeNet's implementation of PKCS#11. Protecttoolkit C represents a suite of products including various PKCS#11 runtimes including software only, hardware adapter, and host security module based variants. A Remote client and server are also available

ProtectToolkit J

SafeNet's implementation of JCE. Runs on top of ProtectToolkit C

R

RC2/RC4

Ciphers designed by RSA Data Security, Inc.

RFC

Request for Comments, proposed specifications for various protocols and algorithms archived by the Internet Engineering Task Force (IETF), see http://www.ietf.org

RNG

Random Number Generator

RSA

Cryptographic algorithm by Ron Rivest, Adi Shamir and Leonard Adelman

RTC

Real Time Clock

S

SDK

Software Development Kits Other documentation may refer to the SafeNet Cprov and Protect Toolkit J SDKs. These SDKs have been renamed ProtectToolkit C and ProtectToolkit J respectively. ☺ The names Cprov and ProtectToolkit C refer to the same device in the context of this or previous manuals. ☺ The names Protect Toolkit J and ProtectToolkit J refer to the same device in the context of this or previous manuals.

Slot

PKCS#11 slot which is capable of holding a token

SlotPKCS#11

Slot which is capable of holding a token

SO

Security Officer

Symmetric Cipher

An encryption algorithm that uses the same key for encryption and decryption. DES, RC4 and IDEA are all symmetric algorithms

T

TC

Trusted Channel

TCP/IP

Transmission Control Protocol / Internet Protocol

Token

PKCS#11 token that provides cryptographic services and access controlled secure key storage

TokenPKCS#11

Token that provides cryptographic services and access controlled secure key storage

U

URI

Universal Resource Identifier

V

VA

Validation Authority

X

X.509

Digital Certificate Standard

X.509 Certificate

Section 3.3.3 of X.509v3 defines a certificate as: "user certificate; public key certificate; certificate: The public keys of a user, together with some other information, rendered unforgeable by encipherment with the private key of the certification authority which issued it"