Advanced Java Application Development for the BlackBerry Smartphone

BlackBerry Academic Program
© 2009 Research In Motion Limited. All rights reserved. BlackBerry®, RIM®, Research In Motion®, SureType®, SurePress™ and related trademarks, names and logos are the property of Research In Motion Limited and are registered and/or used in the U.S. and countries around the world.

Adobe and Acrobat are trademarks of Adobe Systems Incorporated; Apache Tomcat is a trademark of The Apache Software Foundation; Bluetooth is a trademark of Bluetooth SIG; Google and Google Talk are trademarks of Google Inc.; GSM and Global System for Mobile communications are trademarks of the GSM MOU Association; IBM, Domino, Lotus, Lotus Notes, Lotus Organizer and Sametime are trademarks of International Business Machines Corporation; IEEE is a trademark of the Institute of Electrical and Electronics Engineers, Inc.; Java, J2ME, JavaScript, JDBC, and JRE are trademarks of Sun Microsystems, Inc.; Macintosh is a trademark of Apple Inc.; Microsoft, Active Directory, Active X, Entourage, Excel, Internet Explorer, Outlook, Power Point, SQL Server, Visual Studio, Windows, Windows Media, Windows Mobile and Windows Server are trademarks of Microsoft Corporation; Novell and GroupWise are trademarks of Novell, Inc.; PGP is a trademark of PGP Corporation; Plazmic is a trademark of Plazmic Inc.; RSA and RSA SecurID are trademarks of RSA Security; Wi-Fi is a trademark of the Wi-Fi Alliance. All other trademarks are the property of their respective owners.

The BlackBerry smartphone and other devices and/or associated software are protected by copyright, international treaties, and various patents, including one or more of the following U.S. patents: 6,278,442; 6,271,605; 6,219,694; 6,075,470; 6,073,318; D445,428; D433,460; D416,256. Other patents are registered or pending in the U.S. and in various countries around the world. Visit www.rim.com/patents for a list of RIM (as hereinafter defined) patents.

This documentation including all documentation incorporated by reference herein such as those provided or made available by hyperlink is provided or made accessible "AS IS" and "AS AVAILABLE" and with condition, endorsement, guarantee, representation or warranty of any kind by Research In Motion Limited and its affiliated companies ("RIM") and RIM assumes no responsibility for any typographical, technical, or other inaccuracies, errors or omissions in this documentation. In order to protect RIM proprietary and confidential information and/or trade secrets, this documentation may describe some aspects of RIM technology in generalized terms. RIM reserves the right to periodically change information that is contained in this documentation; however, RIM makes no commitment to provide any such changes, updates, enhancements, or other additions to this documentation to you in a timely manner or at all.

This documentation might contain references to third-party sources of information, hardware or software, products or services including components and content such as content protected by copyright and/or third-party web sites (collectively the "Third Party Products and Services"). RIM does not control, and is not responsible for, any Third Party Products and Services including, without limitation the content, accuracy, copyright compliance, compatibility, performance, trustworthiness, legality, decency, links, or any other aspect of Third Party Products and Services. The inclusion of a reference to Third Party Products and Services in this documentation does not imply endorsement by RIM of the Third Party Products and Services or the third party in any way.

EXCEPT TO THE EXTENT SPECIFICALLY PROHIBITED BY APPLICABLE LAW IN YOUR JURISDICTION, ALL CONDITIONS, ENDORSEMENTS, GUARANTEES, REPRESENTATIONS, OR WARRANTIES OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY CONDITIONS, ENDORSEMENTS, GUARANTEES, REPRESENTATIONS OR WARRANTIES OF DURABILITY, FITNESS FOR A PARTICULAR PURPOSE OR USE, MERCHANTABILITY, MERCHANTABILITY QUALITY, NON-INFRINGEMENT, SATISFACTORY QUALITY, OR TITLE, OR ARISING FROM A STATUTE OR CUSTOM OR A COURSE OF DEALING OR USAGE OF TRADE, OR RELATED TO THE DOCUMENTATION OR ITS USE, OR PERFORMANCE OR NON-PERFORMANCE OF ANY SOFTWARE, HARDWARE, SERVICE, OR ANY THIRD PARTY PRODUCTS AND SERVICES REFERENCED HEREIN, ARE HEREBY EXCLUDED. YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY BY STATE OR PROVINCE. SOME JURISDICTIONS MAY NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES AND CONDITIONS. TO THE EXTENT PERMITTED BY LAW, ANY IMPLIED WARRANTIES OR CONDITIONS RELATING TO THE DOCUMENTATION TO THE EXTENT THEY CANNOT BE EXCLUDED AS SET OUT ABOVE, BUT CAN BE LIMITED, ARE HEREBY LIMITED TO NINETY (90) DAYS FROM THE DATE YOU FIRST ACQUIRED THE DOCUMENTATION OR THE ITEM THAT IS THE SUBJECT OF THE CLAIM.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN YOUR JURISDICTION, IN NO EVENT SHALL RIM BE LIABLE FOR ANY TYPE OF DAMAGES RELATED TO THIS DOCUMENTATION OR ITS USE, OR PERFORMANCE OR NON-PERFORMANCE OF ANY SOFTWARE, HARDWARE, SERVICE, OR ANY THIRD PARTY PRODUCTS AND SERVICES REFERENCED HEREIN INCLUDING WITHOUT LIMITATION ANY OF THE FOLLOWING DAMAGES: DIRECT, CONSEQUENTIAL, EXEMPLARY, INCIDENTAL, INDIRECT, SPECIAL, PUNITIVE, OR AGGRAVATED DAMAGES, DAMAGES FOR LOSS OF PROFITS OR REVENUES, FAILURE TO REALIZE ANY EXPECTED SAVINGS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, LOSS OF BUSINESS OPPORTUNITY, OR CORRUPTION OR LOSS OF DATA, FAILURES TO TRANSMIT OR RECEIVE ANY DATA, PROBLEMS ASSOCIATED WITH ANY APPLICATIONS USED IN CONJUNCTION WITH RIM PRODUCTS OR SERVICES, DOWNTIME COSTS, LOSS OF THE USE OF RIM PRODUCTS OR SERVICES OR ANY PORTION THEREOF OR OF ANY AIRTIME SERVICES, COST OF SUBSTITUTE GOODS, COSTS OF COVER, FACILITIES OR SERVICES, COST OF CAPITAL, OR OTHER SIMILAR PECUNIARY LOSSES, WHETHER OR NOT SUCH DAMAGES WERE FORESEEN OR UNFORESEEN, AND EVEN IF RIM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN YOUR JURISDICTION, RIM SHALL HAVE NO OTHER OBLIGATION, DUTY, OR LIABILITY WHATSOEVER IN CONTRACT, TORT, OR OTHERWISE TO YOU INCLUDING ANY LIABILITY FOR NEGLIGENCE OR STRICT LIABILITY.

THE LIMITATIONS, EXCLUSIONS, AND DISCLAIMERS HEREIN SHALL APPLY: (A) IRRESPECTIVE OF THE NATURE OF THE CAUSE OF ACTION, DEMAND, OR ACTION BY YOU INCLUDING BUT NOT LIMITED TO BREACH OF CONTRACT, NEGLIGENCE, TORT, STRICT LIABILITY OR ANY OTHER LEGAL THEORY AND SHALL SURVIVE A FUNDAMENTAL BREACH OR BREACHES OR THE FAILURE OF THE ESSENTIAL PURPOSE OF THIS AGREEMENT OR OF ANY REMEDY CONTAINED HEREIN; AND (B) TO RIM AND ITS AFFILIATED COMPANIES, THEIR SUCCESSORS, ASSIGNS, AGENTS, SUPPLIERS (INCLUDING AIRTIME SERVICE PROVIDERS), AUTHORIZED RIM DISTRIBUTORS (ALSO INCLUDING AIRTIME SERVICE PROVIDERS) AND THEIR RESPECTIVE DIRECTORS, EMPLOYEES AND INDEPENDENT CONTRACTORS.

IN ADDITION TO THE LIMITATIONS AND EXCLUSIONS SET OUT ABOVE, IN NO EVENT SHALL ANY DIRECTOR, EMPLOYEE, AGENT, DISTRIBUTOR, SUPPLIER, INDEPENDENT CONTRACTOR OF RIM OR ANY AFFILIATES OF RIM HAVE ANY LIABILITY ARISING FROM OR RELATED TO THE DOCUMENTATION.
Prior to subscribing for, installing or using any Third Party Products and Services it is your responsibility to ensure that your airtime service provider has agreed to support all of their features. Some airtime service providers may not offer Internet browsing functionality with a subscription to BlackBerry® Internet Service. Check with your service provider for availability, roaming arrangements, service plans and features. Installation or use of Third Party Products and Services with RIM's products and services may require one or more patent, trademark, copyright or other licenses in order to avoid infringement or violation of third party rights. You are solely responsible for determining whether to use, Third Party Products and Services and if any third party licenses are required to do so. If required you are responsible for acquiring them. You should not install or use Third Party Products and Services until all necessary licenses have been acquired. Any Third Party Products and Services that are provided with RIM's products and services are provided as a convenience to you and are provided 'AS IS' with no express or implied conditions, endorsements, guarantees, representations or warranties of any kind by RIM and RIM assumes no liability whatsoever, in relation thereto. Your use of Third Party Products and Services shall be governed by and subject to you agreeing to the terms of separate licenses and other agreements applicable thereto with third parties, except to the extent expressly covered by a license or other agreement with RIM.

Certain features outlined in this documentation require a minimum version of BlackBerry® Enterprise Server, BlackBerry® Desktop Software, BlackBerry® Device Software and/or BlackBerry Handheld Software and may require additional development or Third Party Products and Services for access to corporate applications.

Certain products mentioned in this documentation include software developed by the Apache Software Foundation (http://www.apache.org/) and/or licensed pursuant to Apache License, Version 2.0 (http://www.apache.org/licenses/). For more information, see the NOTICE.txt file included with the software. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

The terms of use of any RIM product or service are set out in a separate license or other agreement with RIM applicable thereto. NOTHING IN THIS DOCUMENTATION IS INTENDED TO SUPERSEDE ANY EXPRESS WRITTEN AGREEMENTS OR WARRANTIES PROVIDED BY RIM FOR PORTIONS OF ANY RIM PRODUCT OR SERVICE OTHER THAN THIS DOCUMENTATION.
# Chapter 9

## Developing secure applications

### Objectives

- Define cryptography
- Define authentication
- Discuss the RIM® Cryptographic API
- Use encryption in an application
- Use BlackBerry® Signing Authority Tool

This chapter outlines security concerns related to Java® applications for BlackBerry® smartphones. It defines cryptography, and describes the steps you can take to encrypt communication and authenticate users, including the steps you need to take to manage certificates.
Cryptography and encryption

Cryptography is the science of information security; cryptographers study ways to hide information in storage or transit.

In computing, cryptography is most often associated with software that can scramble (encrypt) plaintext (ordinary text, sometimes referred to as cleartext) into ciphertext, then unscramble (decrypt) it back into clear text. Modern cryptography concerns itself with the following three goals:

• Confidentiality—ensure that information can be understood only by the intended recipient.
• Integrity—ensure that information cannot be altered in storage or transit between sender and intended recipient without the alteration being detected.
• Authentication—ensure that the sender and recipient can each confirm the other’s identity, and that each can confirm the origin/destination of the information.

In addition, cryptography can ensure nonrepudiation of information, ensuring that the sender of information cannot later deny that they created or transmitted the information.

Goals of cryptography in digital systems

Cryptography is important to the security and integrity of transmitted data, but no protocol is entirely secure. Protocols and algorithms are constantly being assaulted by increasingly skilled attackers with increasingly more powerful computers. At the same time, cryptographers are constantly improving routines and algorithms by increasing key sizes and thereby greatly increasing the effort required to compromise security.

The three main goals of cryptography are as follows:

• Confidentiality

The most common use of cryptography in the business environment is to provide data encryption. Encryption is the act of disguising a message in such a way that its true meaning is shielded. Inherent is the concept of encryption is the idea of decryption; the intended recipient must be able to recover the information.

Many different ways exist to encrypt and decrypt information, but the most common approach is through the use of ciphers. A plaintext message is encoded using a predetermined and agreed upon protocol and cipher. The resulting ciphertext message is then transmitted to the recipient, who then decrypts the message using the agreed upon protocol.

Many different encryption protocols exist. Some are more secure or more practical than others.

• Integrity
If an encryption protocol is compromised, an intermediary can potentially read, delete, or modify your data, or even intercept the messages and pretend to be the legitimate owner of the information.

Data integrity is achieved in modern cryptography by using a hash function to produce a unique digital fingerprint of a document. To achieve this, the sender applies a complex hash function to a document to create a unique value. When the message is delivered, the recipient applies the same hash function to the message. If the resulting values match, then the recipient can have a high degree of confidence that the message has not been modified.

A variety of hashing routines exist for use in different scenarios. A common hashing routine is the MAC. MACs, which are described in greater detail later in this document, combine encryption keys and hashing functions to allow users to transmit highly secure, key-dependent hash values.

- **Authentication**

  In the event that both the hash function and the encryption protocol are compromised, an intermediary user can both read confidential data and impersonate a legitimate user. Sender authentication can help to protect against this. To achieve this, authentication protocols use a digital signature to electronically sign a document to verify the identity of the sender.

  One common protocol combines a digital signature with a private key encryption routine to create a type of digital stamp. To decrypt the message, the recipient must decrypt the digital stamp using the sender’s private key. As long as the sender’s private key remains secure, this method assures the authenticity of the digital signature.

  Digital certificates are electronic documents which bind a public key to an identity, verifying that a public key belongs to an entity, such as an individual or company. Digital certificates are often used by software companies to distribute applications over the Internet.
1. Modern cryptography concerns itself with which of the following objectives?
   A. Confidentiality--ensures that information cannot be understood by anyone other than the party for whom it is intended.
   B. Integrity--ensures that information cannot be altered in storage or transit between sender and intended recipient without the alteration being detected.
   C. Nonrepudiation--ensures that the creator/sender of information cannot deny at a later stage his or her intentions in the creation or transmission of the information.
   D. Authentication--ensures that the sender and recipient can confirm each other’s identity and the origin/destination of the information.
   E. All of the above
   F. None of the above

2. Which of the following is true?
   A. Cryptosystems include mathematical procedures, computer programs, but do not concern themselves with the regulation of human behavior.
   B. Cryptosystems include mathematical procedures, computer programs, and the regulation of human behavior.
   C. Cryptosystems are not used in modern computing.

3. Which of the following is true?
   A. Only one encryption protocol is in use in modern computing.
   B. Many different encryption protocols exist.
   C. Two encryption protocols exist, and only these two are possible (MAC and hash).
Answers

1. E
2. B
3. B
RIM Cryptographic API

The RIM Cryptographic API is a collection of classes that allows you to provide security for your BlackBerry application. The API is robust and flexible so that you can accomplish each task in a number of different ways, depending on the needs of your application.

Use the RIM Cryptographic API to complete the following tasks:
- encrypt and decrypt data
- digitally sign and verify data (secure the integrity of your data)
- authenticate data

Did you know

The CLDC and MIDP of J2ME do not define a cryptographic API. The RIM Cryptographic API does not follow the java.security model for the following reasons:
- Multiple cryptographic algorithm providers are not required in embedded development.
- Compile time checking simplifies embedded development, compared to runtime checking.

Controlling access to APIs and application data

RIM tracks the use of sensitive APIs in the BlackBerry® Java® Development Environment for security and export control reasons.

Use the API reference to determine whether the BlackBerry Java Plug-in for Eclipse has a lock icon, or is designated as signed. If either of these is the case, then your BlackBerry smartphone application requires a signed key or signature, which RIM provides, before you can load the BlackBerry smartphone application COD files onto a BlackBerry smartphone.

RIM controls Runtime APIs, BlackBerry Application APIs, and BlackBerry Cryptography APIs. Without the code signatures, you can test BlackBerry smartphone applications that use controlled APIs in the BlackBerry Smartphone Simulator. However, you must obtain code signatures from RIM before you can load the BlackBerry smartphone applications onto BlackBerry smartphones.

If you use any of the following BlackBerry API packages, your BlackBerry smartphone application requires code signatures before you can load it onto a BlackBerry smartphone:
- net.rim.blackberry.api.browser
- net.rim.blackberry.api.invoke
- net.rim.blackberry.api.mail
- net.rim.blackberry.api.mail.event
Developing secure applications

- net.rim.blackberry.api.menuitem
- net.rim.blackberry.api.options
- net.rim.blackberry.api.pdap
- net.rim.blackberry.api.phone
- net.rim.blackberry.api.phone.phonelogs
- net.rim.device.api.browser.field
- net.rim.device.api.browser.plugin
- net.rim.device.api.crypto
- net.rim.device.api.io.http
- net.rim.device.api.notification
- net.rim.device.api.servicebook
- net.rim.device.api.synchronization
- net.rim.device.api.system

Registering to use controlled APIs

To use controlled APIs, you must first register with RIM.


2. Save the .csi file that RIM sends to you. The .csi file contains a list of signatures and your registration information.

   If the BlackBerry Signing Authority Tool administrator does not provide you with the .csi file or the Client PIN contact your ISV Technical Partnership Manager, if you are an ISV partner, or send an email message to jde@rim.com, if you are not an ISV partner.

3. Double-click the .csi file.

4. If a dialog box appears stating that a private key cannot be found, complete steps 5 to 8 before you continue. Otherwise, proceed to step 9.

5. Click Yes to create a new key pair file.

6. In the Private Key Password field, type a password of at least eight characters, and type it again to confirm. The private key password protects your private key. If you lose this password, you must register again with RIM. If this password is stolen, contact RIM immediately.

7. Click OK.

8. Move your mouse to generate data for a new private key.

9. In the Registration PIN field, type the PIN that RIM provided.
10. In the **Private Key Password** field, type the **private key** password.

11. Click **Register**.

12. Click **Exit**.

For more information about signing your code to get access to the cryptography APIs, see *BlackBerry Signing Authority Tool*, later in this chapter.

### Components of the RIM Cryptographic API

The RIM Cryptographic API consists of several interrelated APIs.

- The Secure Messaging and Secure Connection APIs contain the code necessary for implementing secure communication, and therefore they define the protocol functionality.
  - The Secure Messaging API contains the **CMS** API and provides the functionality needed to create a secure messaging application.
  - The Secure Connection API contains the **TLS**, **WTLS**, and **SSL** APIs and provides the functionality required to create and manage secure connections between client and server.

<table>
<thead>
<tr>
<th>Secure Messaging API</th>
<th>Secure Connection API</th>
</tr>
</thead>
<tbody>
<tr>
<td>• CMS API</td>
<td>• TLS API</td>
</tr>
<tr>
<td></td>
<td>• SSL API</td>
</tr>
<tr>
<td></td>
<td>• WTLS API</td>
</tr>
</tbody>
</table>

- The Key Management API contains the basic cryptographic framework needed to create secure applications.
  - The KeyStore API handles key management and distribution.
  - The Encoding API handles key encoding.
  - The Certificate API contains the functionality necessary for managing cryptographic certificates.
  - The **ASN.1** API provides a mechanism to format and parse data that is commonly needed with cryptographic schemes and protocols.
• The **OID** API contains the functionality necessary for handling and using some popular Object IDs.

<table>
<thead>
<tr>
<th>Key Management API</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyStore API</td>
</tr>
<tr>
<td>- KeyStore</td>
</tr>
<tr>
<td>- DesktopKeyStore</td>
</tr>
<tr>
<td>- SyncableRIMKeyStore</td>
</tr>
<tr>
<td>- RIMKeyStore</td>
</tr>
<tr>
<td>- PersistableRIMKeyStore</td>
</tr>
<tr>
<td>- TrustedKeyStore</td>
</tr>
<tr>
<td>Encoding API</td>
</tr>
<tr>
<td>- Private Key Encoders</td>
</tr>
<tr>
<td>- Public Key Encoders</td>
</tr>
<tr>
<td>- Signature Encoders</td>
</tr>
<tr>
<td>- Symmetric Key Encoders</td>
</tr>
<tr>
<td>Certificate API</td>
</tr>
<tr>
<td>- Certificate</td>
</tr>
<tr>
<td>- X509</td>
</tr>
<tr>
<td>- Status</td>
</tr>
<tr>
<td>- WTLS</td>
</tr>
<tr>
<td>ASN.1 API</td>
</tr>
<tr>
<td>- ASN1InputStream</td>
</tr>
<tr>
<td>- ASN1OutputStream</td>
</tr>
<tr>
<td>- ASN1InputByteArray</td>
</tr>
<tr>
<td>OID API</td>
</tr>
<tr>
<td>- OID</td>
</tr>
<tr>
<td>- OIDs</td>
</tr>
</tbody>
</table>

• The Cryptographic Primitives API contains the most basic tools needed to implement the cryptographic utilities.

• The Cryptographic Primitives API contains the keys, MACs, ciphers and other functionality associated with both symmetric and public cryptography.

• The Cryptographic Primitives API also contains the unkeyed algorithms such as digests and PRNGs needed by other members in the API.

<table>
<thead>
<tr>
<th>Cryptographic Primitives API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric Key Algorithms</td>
</tr>
<tr>
<td>Public Key Algorithms</td>
</tr>
<tr>
<td>Miscellaneous</td>
</tr>
<tr>
<td>- Digests PRNGs</td>
</tr>
</tbody>
</table>
## Supported algorithms

### Symmetric Key Algorithms

<table>
<thead>
<tr>
<th>Keys</th>
<th>Encryptors Algorithms</th>
<th>Decryptors Algorithms</th>
<th>MACs</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>AES ARC4</td>
<td>AES ARC4</td>
<td>CBCMAC, HMAC</td>
</tr>
<tr>
<td>ARC4</td>
<td>DES</td>
<td>DES</td>
<td></td>
</tr>
<tr>
<td>Cast128</td>
<td>TripleDES</td>
<td>TripleDES</td>
<td></td>
</tr>
<tr>
<td>DES</td>
<td>RC2</td>
<td>RC2</td>
<td></td>
</tr>
<tr>
<td>TripleDES</td>
<td>Skipjack CAST128</td>
<td>Skipjack CAST128</td>
<td></td>
</tr>
<tr>
<td>RC2</td>
<td>CBC CFB</td>
<td>CBC CFB</td>
<td></td>
</tr>
<tr>
<td>RC5</td>
<td>OFB X</td>
<td>OFB X</td>
<td></td>
</tr>
<tr>
<td>Skipjack</td>
<td>PKCS5</td>
<td>PKCS5</td>
<td></td>
</tr>
<tr>
<td>HMAC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Public Key Algorithms

<table>
<thead>
<tr>
<th>Keys</th>
<th>Key Agreement</th>
<th>Encryptors</th>
<th>Decryptors</th>
<th>Signers</th>
<th>Verifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>DH</td>
<td>DH</td>
<td>RSA PKCS1</td>
<td>RSA PKCS1</td>
<td>DSA</td>
<td>DSA</td>
</tr>
<tr>
<td>DSA</td>
<td>ECDH</td>
<td>RSA OAEP</td>
<td>RSA OAEP</td>
<td>ECDSA</td>
<td>ECDSA</td>
</tr>
<tr>
<td>EC</td>
<td>ECMQV</td>
<td>ElGamal</td>
<td>ElGamal</td>
<td>ECNR</td>
<td>ECNR</td>
</tr>
<tr>
<td>KEA</td>
<td>KEA</td>
<td></td>
<td></td>
<td>RSA PKCS1</td>
<td>RSA PKCS1</td>
</tr>
<tr>
<td>RSA</td>
<td>KDFs, P1363,</td>
<td></td>
<td></td>
<td>RSA PSS</td>
<td>RSA PSS</td>
</tr>
<tr>
<td></td>
<td>X9.42 KDF</td>
<td></td>
<td></td>
<td>X9.31</td>
<td>X9.31</td>
</tr>
</tbody>
</table>
Developing secure applications

Extending the RIM Cryptographic API

The RIM Cryptographic API employs a *factory* class of flexible cryptography algorithms that you can implement in a number of different ways; this means that you can use the API to provide security, regardless of the size or complexity of your project.

The RIM Cryptographic API contains a variety of encryption algorithms that you can implement individually as low-level algorithms, or as part of a larger, more customized process. You can also extend the API by implementing your own cryptographic algorithms, encoding schemes, certificates and software tokens.

Keys, CryptoSystems, Certificates and KeyStores

This section describes keys, and the technology associated with them.

Keys

In the RIM Cryptographic API, Key is an interface. Subinterfaces for various different types of keys extend from the Key interface, including the following subinterfaces:

- **Public Key**—the public key is a key that can have its contents divulged to anyone. Public keys are used in *asymmetric key* cryptography, where the public key and private key are different.
- **Private Key**—the private key contains all of the secret information in asymmetric key cryptography. You must protect the content of this key.
- **Symmetric Key**—in symmetric key cryptography, a single key is used for both encrypting and decrypting. You must protect the content of this key.

<table>
<thead>
<tr>
<th><strong>Miscellaneous</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Digests</strong></td>
<td><strong>PRNGs</strong></td>
</tr>
<tr>
<td>• SHA1</td>
<td>• P1363 KDF1</td>
</tr>
<tr>
<td>• SHA256</td>
<td>• PKCS1 MGF1</td>
</tr>
<tr>
<td>• SHA384</td>
<td>• X9.42 KDF</td>
</tr>
<tr>
<td>• SHA512</td>
<td>• PKCS5 KDF1</td>
</tr>
<tr>
<td>• MD2</td>
<td>• PKCS5 KDF2</td>
</tr>
<tr>
<td>• MD4</td>
<td>• FIPS186 PRNG</td>
</tr>
<tr>
<td>• MD5</td>
<td>• RFC 2631 KDF</td>
</tr>
<tr>
<td>• RIPEMD128</td>
<td></td>
</tr>
<tr>
<td>• RIPEMD160</td>
<td></td>
</tr>
</tbody>
</table>
Sample code: creating a DES key

```java
// uses random data to create the key.
DESKey unknownKey = new DESKey();

// data contains the known key.
byte[] data = new byte[ 8 ];
DESKey knownKey = new DESKey( data );
```

CryptoSystems

Public key cryptographic algorithms (including RSA, DSA and ECDSA) use cryptosystems to hold public information common to all users within a cryptographic community or system. The cryptosystem typically uses numerical values that each party must possess for encrypted communication to succeed.

Cryptosystems provide the tools you need to create and use keys, while allowing you to specify various parameters for the keys. Each of the following cryptosystems has unique attributes:

- **RSACryptoSystem**
- **DSACryptoSystem**
- **ECCryptoSystem**
- **DHCryptoSystem**

RSACryptoSystem

The RSACryptoSystem is the easiest-to-use of the supported CryptoSystems, because it requires only that you enter the size of the modulus.

<table>
<thead>
<tr>
<th>Result</th>
<th>Sample code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create an RSA keypair.</td>
<td>RSACryptoSystem rsaCryptoSystem = new RSACryptoSystem(1024);</td>
</tr>
<tr>
<td>Retrieve the keys.</td>
<td>RSAPublicKey rsaPublicKey = rsaKeyPair.getKey();</td>
</tr>
<tr>
<td></td>
<td>RSAPublicKey rsaPublicKey = rsaKeyPair.getPublicKey();</td>
</tr>
<tr>
<td></td>
<td>RSAPrivateKey rsaPrivateKey = rsaKeyPair.getPrivateKey();</td>
</tr>
<tr>
<td>Create a keypair with known n and e values.</td>
<td>// A key with modulus 1024 bits in length.</td>
</tr>
<tr>
<td></td>
<td>RSACryptoSystem rsaCryptoSystem = new RSACryptoSystem(1024);</td>
</tr>
<tr>
<td></td>
<td>// where n and e are byte arrays</td>
</tr>
<tr>
<td></td>
<td>RSAKeyUp rsaPublicKey = new RSAKeyUp(rsaCryptoSystem, e, n );</td>
</tr>
</tbody>
</table>
For an RSA public key, the n and e values are stored in a byte array format, which provides flexibility, since most classes can be converted into a byte array.

**DSACryptoSystem**

The DSACryptoSystem relies on system parameters that are predefined, and so are not included in each message request. This reduces the size of the message request. You can use the DSACryptoSystem to store constant parameters representing P, Q, and G.

Two constructors are associated with the DSACryptoSystem.

The first constructor uses the default parameters (which are shown in the following code sample as byte arrays) and hides many of the parameters:

```java
DSACryptoSystem system = new DSACryptoSystem();
```

The second constructor allows you to specify the parameters. In the following code sample, the constructor is called with the default values for p, q, and g:

```java
/* 1024-bit key parameters */
private static final byte[] p = new byte[] {
    (byte)0xfd, (byte)0x7f, (byte)0x53,(byte)0x81, (byte)0x1d,
    (byte)0x75, (byte)0x12, (byte)0x29,
    (byte)0x52, (byte)0xdf,(byte)0x4a, (byte)0x9c, (byte)0x2e,
    (byte)0xec, (byte)0xe4,(byte)0xe7,
    (byte)0xf6, (byte)0x11, (byte)0xb7, (byte)0x52, (byte)0x3e,
    (byte)0xef, (byte)0x44, (byte)0x00,
    (byte)0xc3, (byte)0x1e, (byte)0x3f, (byte)0x80, (byte)0xb6,
    (byte)0x51, (byte)0x26, (byte)0x69,
    (byte)0x45, (byte)0x5d, (byte)0x40, (byte)0x22, (byte)0x51,
    (byte)0xf6, (byte)0x59, (byte)0x3d,
    (byte)0x8d, (byte)0x58, (byte)0xfa, (byte)0xbf, (byte)0xc5,
    (byte)0xf5, (byte)0x83, (byte)0x37, (byte)0x30,
    (byte)0xf6, (byte)0xcb, (byte)0x9b, (byte)0x55, (byte)0x6c,
    (byte)0xe7, (byte)0x81, (byte)0x3b,
    (byte)0x8a, (byte)0x1d, (byte)0x3d, (byte)0x9f, (byte)0xe8,
    (byte)0x66, (byte)0xb7,
    (byte)0x69, (byte)0x99, (byte)0x50, (byte)0x50, (byte)0xa5,
    (byte)0xa4, (byte)0xe8,
};
```
private static final byte[] q = new byte[] {
    (byte)0x97, (byte)0x60, (byte)0x50, (byte)0x8f, (byte)0x15,
    (byte)0x23, (byte)0x0b, (byte)0xcc,
    (byte)0xb2, (byte)0x92, (byte)0xb9, (byte)0x82, (byte)0xa2,
    (byte)0xeb, (byte)0x84, (byte)0x0b,
    (byte)0xf0, (byte)0x58, (byte)0x1c, (byte)0xf5
};

private static final byte[] g = new byte[] {
    (byte)0xf7, (byte)0xe1, (byte)0xa0, (byte)0x85, (byte)0xd6,
    (byte)0x9b, (byte)0x3d, (byte)0xde,
    (byte)0xcb, (byte)0xbc, (byte)0xab, (byte)0x5c, (byte)0x36,
    (byte)0xb8, (byte)0x57, (byte)0xb9,
    (byte)0x79, (byte)0x94, (byte)0xaf, (byte)0xbb, (byte)0xfa,
    (byte)0x3a, (byte)0xea, (byte)0x82,
(byte)0xf9, (byte)0x57, (byte)0x4c, (byte)0x0b, (byte)0x3d,
(byte)0x07, (byte)0x82, (byte)0x49,
(byte)0x51, (byte)0x59, (byte)0x57, (byte)0x8e, (byte)0xba,
(byte)0xd4, (byte)0x59, (byte)0x4f,
(byte)0xe6, (byte)0x71, (byte)0x07, (byte)0x10, (byte)0x81,
(byte)0x80, (byte)0xb4, (byte)0x49,
(byte)0x16, (byte)0x71, (byte)0x23, (byte)0xe8, (byte)0x4c,
(byte)0x28, (byte)0xa6, (byte)0xe1,
(byte)0x3c, (byte)0x16, (byte)0x7a, (byte)0x8b, (byte)0x54,
(byte)0x7c, (byte)0x8d, (byte)0xe0, (byte)0xa3, (byte)0xae,
(byte)0x91, (byte)0x6e, (byte)0xa3, (byte)0x7f, (byte)0x0b,
(byte)0xfa, (byte)0x35, (byte)0x62, (byte)0xf1, (byte)0xfb,
(byte)0x01, (byte)0x24, (byte)0x3b,
(byte)0xcc, (byte)0xa4, (byte)0xf1, (byte)0xfb, (byte)0x0f,
(byte)0x0a, (byte)0x90, (byte)0x0b, (byte)0x0a,
(byte)0x51, (byte)0x90, (byte)0x89,
(byte)0xa8, (byte)0x83, (byte)0xdf, (byte)0xe1, (byte)0x5a,
(byte)0xe5, (byte)0x9f, (byte)0x06,
(byte)0x92, (byte)0x8b, (byte)0x66, (byte)0x5e, (byte)0x80,
(byte)0x7b, (byte)0x55, (byte)0x25,
(byte)0x64, (byte)0x01, (byte)0x4c, (byte)0x3b, (byte)0xfe,
(byte)0xc8, (byte)0xa6, (byte)0xe1,
(byte)0x92, (byte)0x8b, (byte)0x66, (byte)0x5e, (byte)0x80,
(byte)0x7b, (byte)0x55, (byte)0x25,
(byte)0x64, (byte)0x01, (byte)0x4c, (byte)0x3b, (byte)0xfe,
(byte)0xc8, (byte)0xa6, (byte)0xe1,
(byte)0x92, (byte)0x8b, (byte)0x66, (byte)0x5e, (byte)0x80,
(byte)0x7b, (byte)0x55, (byte)0x25,
(byte)0x64, (byte)0x01, (byte)0x4c, (byte)0x3b, (byte)0xfe,
(byte)0xc8, (byte)0xa6, (byte)0xe1,
(byte)0x92, (byte)0x8b, (byte)0x66, (byte)0x5e, (byte)0x80,
(byte)0x7b, (byte)0x55, (byte)0x25,
(byte)0x64, (byte)0x01, (byte)0x4c, (byte)0x3b, (byte)0xfe,
(byte)0xc8, (byte)0xa6, (byte)0xe1,
(byte)0x92, (byte)0x8b, (byte)0x66, (byte)0x5e, (byte)0x80,
(byte)0x7b, (byte)0x55, (byte)0x25,
(byte)0x64, (byte)0x01, (byte)0x4c, (byte)0x3b, (byte)0xfe,
(byte)0xc8, (byte)0xa6, (byte)0xe1,
(byte)0x92, (byte)0x8b, (byte)0x66, (byte)0x5e, (byte)0x80,
(byte)0x7b, (byte)0x55, (byte)0x25,
(byte)0x64, (byte)0x01, (byte)0x4c, (byte)0x3b, (byte)0xfe,
(byte)0xc8, (byte)0xa6, (byte)0xe1,
(byte)0x92, (byte)0x8b, (byte)0x66, (byte)0x5e, (byte)0x80,
(byte)0x7b, (byte)0x55, (byte)0x25,
(byte)0x64, (byte)0x01, (byte)0x4c, (byte)0x3b, (byte)0xfe,
(byte)0xc8, (byte)0xa6, (byte)0xe1,
(byte)0x92, (byte)0x8b, (byte)0x66, (byte)0x5e, (byte)0x80,
(byte)0x7b, (byte)0x55, (byte)0x25,
(byte)0x64, (byte)0x01, (byte)0x4c, (byte)0x3b, (byte)0xfe,
(byte)0xc8, (byte)0xa6, (byte)0xe1,
(byte)0x92, (byte)0x8b, (byte)0x66, (byte)0x5e, (byte)0x80,
(byte)0x7b, (byte)0x55, (byte)0x25,
(byte)0x64, (byte)0x01, (byte)0x4c, (byte)0x3b, (byte)0xfe,
(byte)0xc8, (byte)0xa6, (byte)0xe1,
(byte)0x92, (byte)0x8b, (byte)0x66, (byte)0x5e, (byte)0x80,
(byte)0x7b, (byte)0x55, (byte)0x25,
(byte)0x64, (byte)0x01, (byte)0x4c, (byte)0x3b, (byte)0xfe,
(byte)0xc8, (byte)0xa6, (byte)0xe1,
(byte)0x92, (byte)0x8b, (byte)0x66, (byte)0x5e, (byte)0x80,
(byte)0x7b, (byte)0x55, (byte)0x25,
(byte)0x64, (byte)0x01, (byte)0x4c, (byte)0x3b, (byte)0xfe,
(byte)0xc8, (byte)0xa6, (byte)0xe1,
Certificates

Digital certificates are electronic documents which bind a public key to an identity, verifying that a public key belongs to an entity, such as an individual or company. The entity can then use the certificate as a way of providing the public key to other parties. Certificates are signed by a third party called a CA. The CA is trusted and, since they sign the certificate, the certificate is trusted. Someone wishing to use a certificate can verify the signature to ensure that the key it provides is legitimate.

Various standards exist for certificates, the most common of which are X509 and WTLS. Each type of certificate has specific features, but in all cases you can manage them using the Certificate Interface.

Certificate Interface

The following options are available through the Certificate Interface:

- If you have the encoding for a certificate, you can use the following code sample to create an X509 certificate; replace the text encoding with an appropriate string:

```java
byte[] encoding = encoding;
X509Certificate certificate = new X509Certificate(encoding);
```

You can also create a certificate from its encoding using a Certificate Factory.

```java
InputStream encoding = encoding;
Certificate certificate = CertificateFactory.getInstance("X509", encoding);
```

The Certificate Factory is able to read X509 and WTLS Certificates. If you want to use another type of certificate, you must register it with the Certificate Factory.

To create a new X509 Certificate, use the X509Certificate.createCertificate() call. For example, the following code sample creates a root certificate (a certificate where the issuer and subject are the same):

```java
RSAKeyPair keyPair = new RSAKeyPair(new RSACryptoSystem());
X509DistinguishedName name = new X509DistinguishedName("O=ACME Corp., C=Canada");
long keyUsage = KeyUsage.KEY_CERT_SIGN;
byte[] serialNumber = new byte[] { (byte)0x01 };
long validNotBefore = System.currentTimeMillis();
long validNotAfter = validNotBefore + 1000*60*60*24*365;// add a year of milliseconds
X509Certificate root = X509Certificate.createX509Certificate(keyPair, name, keyUsage, serialNumber, null, validNotBefore, validNotAfter);
```

- The equals() method compares the fields of two certificates.
- The getEncoding() method returns the encoded format of the certificate.
• The getPublicKey() method extracts the public key associated with the certificate (every certificate must have a public key associated with it).
• The verify() method verifies the validity of a certificate.
• The certificate API also supports the concept of certificate chains. You can create a certificate chain using any of several methods.
  • If you have all the certificates as individual objects, then you can put them in an array.

```java
Certificate endEntity = ...
Certificate firstCA = ...
Certificate rootCA = ...
Certificate[] chain = new Certificate[] { endEntity, firstCA, rootCA };
```

• Alternatively, you can use the build certificate chain utility, found in CertificateUtilities. This example assumes that one of the CAs is in an array of certificates, and the other CA is in a key store.

```java
Certificate endEntity = ...
Certificate[] pool = ... (contains firstCA)
KeyStore keyStore = ... (contains rootCA)
Certificate[] chain = CertificateUtilities.buildCertChain
    endEntity, pool, keyStore);
```

Use the following code sample to determine whether the chain is trusted (that is, whether you trust one of the certificates in the chain).

```java
KeyStore trustedKeyStore = TrustedKeyStore.getInstance();
boolean trusted = CertificateUtilities.isCertificateChainTrusted( chain, trustedKeyStore )
```

• You can create a trusted certificate chain directly.

```java
Certificate endEntity = ...
Certificate[] pool = ... (contains firstCA)
KeyStore keyStore = ... (contains rootCA)
KeyStore trustedKeyStore = TrustedKeyStore.getInstance();
Certificate[] trustedChain = CertificateUtilities.buildTrustedCertChain( endEntity, pool, keyStore, trustedKeyStore );
```

• Each certificate on a BlackBerry smartphone has a CertificateStatus. The CertificateStatus indicates if the status of the certificate is good, revoked, or unknown. Status values are managed by the CertificateStatusManager.

### Keystores

A **keystore** is a class that you can use to simplify the storage of keys on a BlackBerry smartphone.

The keystore classes are made up of three components.
• The KeyStore interface sets the stage for any other keystore that is to be implemented for the RIM Cryptographic APIs. It provides functions that you can use to store and retrieve keys and certificates.
• The RIMKeyStore class is provided as a default implementation.
• The TrustedKeyStore class provides secure storage for trusted certificates and public keys without the possibility of tampering. The public interface to the class provides get methods to retrieve the certificates and public keys.
Root certificates for all of the CAs are preloaded into the browser before shipping, so BlackBerry smartphone users can trust the root certificates. These trusted root certificates form the basis for all subsequent chains of trust.

Encrypted and decrypting data

This section describes the RIM Cryptographic API, and provides detailed examples of how to use the algorithms contained within it.

The RIM Cryptographic API is a flexible development framework that you can use to accomplish a given task using various levels of abstraction. For example, you can perform either of the following:
• You can encrypt a block of data at the lowest level of abstraction using a simple encryption algorithm. You are not required to specify values for the encryption operation.
• You can use a factory class, for example the EncryptorFactory, to encode a stream of data while specifying all details of the encryption itself. You must provide several arguments, including a string representing the encryption algorithm, an object representing the input, the OutputStream, and an randomizing initialization.

The following code samples illustrate the various levels of abstraction provided within the API, by using different processes to accomplish a given task.

Consider the needs of your application (and your level of experience with Java development) in determining which method to use. Developers who are comfortable with cryptographic techniques and who need to further customize their software solution can use a low-level encryption routine (such as the TripleDESEncryptorEngine), whereas developers with little or no cryptographic experience can choose to implement their solution using the EncryptorFactory class.

Block ciphers

This section discusses engines, encryptors, and decryptors.

A block cipher operates on fixed-length groups of bits, (blocks), and applies the same transformation to each one. The block of cipher text a block cipher produces is the same length as the plaintext it encrypts.
Engines

Engines are provided to allow you access to the lowest level of abstraction. For example, you can directly access the DES algorithm by passing in a plaintext byte array, and receive an encrypted byte array using the DESEncryptorEngine. You can do all your work at this level, or you can choose to work at a higher level using the EncryptorFactory, which is built upon the engines. Both methods for encrypting and decrypting produce identical results.

Encryptors and decryptors

The idea behind the encryptors and decryptors is to add a stream-based method (built upon the Java InputStream and OutputStream) for performing cryptographic operations. The encryptors and decryptors simplify your work, allowing you to use the common read and write routines instead of low-level engine functions. The interface for read and write is similar to that of the Java API.

Encryption with block ciphers

Several options exist for encrypting using DES. Base your choice of mode (such as ECB, OFB, CFB, or CBC) on the level of security your application requires, and the relative ease-of-use and capabilities of each mode.

Supported modes

The RIM Cryptographic APIs supports the following standard modes for block encryption:

- ECB—(default for all Engine classes)
- Cipher Block Chaining (CBC)
- Cipher Feedback (CFB)
- Output Feedback (OFB)

You can use CFB and OFB to convert a block cipher to a stream cipher. For more information, see the Stream Cipher section.

Initialization Vectors

An IV is a string of random-seeming information that is used with some block cipher modes to ensure that a unique ciphertext message appears each time a given plaintext message is encrypted.
Understanding the sample application, cryptodemo.java

The following example, which is provided in its entirety in the BlackBerry Java Plug-in for Eclipse, illustrates the use of the Triple-DES encryption algorithm. The project, called cryptodemo, encrypts and decrypts a string of text and compares the resulting text string with the original input.

Note:

To access the RIM Cryptographic APIs, you must include the following Import statement in the header of your class file:

```java
import net.rim.device.api.crypto.*;
```

This line of code imports the crypto library, and allows your application to access the classes within it.

Sample encryption using DES and a block cipher in ECB mode

All Engines operate in ECB mode, which means they take in one block at a time and return one block at a time. There is no chaining or dependence on other blocks for the encryption of the ciphertext. ECB is not the most secure mode for encryption; if you require higher security, consider using CBC or one of the other supported modes.

The following sample code uses the DESEncryptorEngine:

```java
// Input
byte[] input = { 'T','e','s','t','i','n','g','!' };

// Output
byte[] output = new byte[ 8 ];
byte[] keyData = { (byte)0x01, (byte)0x23, (byte)0x45,(byte)0x67,
(byte)0x89, (byte)0x01, (byte)0x23, (byte)0x45 };

// Create a new DES key with the given data.
DESKey key = new DESKey( keyData );

// Create engine.
DESEncryptorEngine engine = new DESEncryptorEngine( key );

// Encrypt the input with offset of zero.
engine.encrypt( input, 0, output, 0);
System.out.println("Ciphertext = " + new String( output ) + ".");
```
Sample encryption using DES and a stream cipher in CBC mode

The following sample code uses a BlockEncryptor to encrypt a stream of data with DES, and uses CBC mode:

```java
//Input
byte[] input = { (byte)'T', (byte)'h', (byte)'i', (byte)'s',
(byte)' ', (byte)'i', (byte)'s', (byte)' ', (byte)'a', (byte)'t',
(byte)'e', (byte)'s', (byte)'t', (byte)'!', (byte)'!'};
byte[] iv = { (byte)0xFF, (byte)0xFE, (byte)0xFD, (byte)0xFC,
(byte)0xFB, (byte)0xFA, (byte)0xF9, (byte)0xF8 };
byte[] keyData = { (byte)0x01, (byte)0x23, (byte)0x45,
(byte)0x67, (byte)0x89, (byte)0x01, (byte)0x23, (byte)0x45 };  
// Create a new DES key with the given data.
DESKey key = new DESKey( keyData );
// Create the DES Engine.
DESEncryptorEngine desEngine = new DESEncryptorEngine( key );
// Create the CBC engine.
CBCEncryptorEngine cbcEngine = new CBCEncryptorEngine( desEngine,
new InitializationVector( iv ) );
// Create a stream to hold the encrypted data.
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
// Now we create the Encryptor using the CBCEncryptorEngine and
// the output stream.
BlockEncryptor encryptor = new BlockEncryptor( cbcEngine, output-
Stream );
// Finally, we can write out the data to encrypt.
encryptor.write( input, 0, input.length );
// We want to close the encryptor and grab the bytes.
encryptor.close();
byte[] output = outputStream.toByteArray();
System.out.println("Ciphertext = " + new String( output ) + ").");
```
This approach requires more lines of code than a block cipher, but it yields clearer and more reliable results. The actual encryption is performed with the call to write. Every other line in the sample sets up or tears down the encryptors and engines.

The stream-based approach is useful when you are encrypting data that is already in a stream (rather than in a byte array). For example, it is more efficient to encrypt an HTTP stream using stream-based encryption than it is to convert the stream into a byte array, encrypt it, and then convert it back into a stream.

Decryption with block ciphers

This section discusses block cipher decryption, and the use of formatters and unformatters.

Sample decryption using block ciphers

You can use the DESDecryptorEngine to perform decryption.

```java
// Input
byte[] input = { (byte)0x77, (byte)0xFB, (byte)0xF4, (byte)0x94, (byte)0xE9, (byte)0x70, (byte)0xDD, (byte)0x0B };

// Output
byte[] output = new byte[ 8 ];

// Key Data from encryption
byte[] keyData = { (byte)0x01, (byte)0x23, (byte)0x45, (byte)0x67, (byte)0x89, (byte)0x01, (byte)0x23, (byte)0x45 };

// Create a new DES key with the given data.
DESKey key = new DESKey( keyData );

// Create engine.
DESDecryptorEngine engine = new DESDecryptorEngine( key );

// Decrypt the input with offset of zero.
engine.decrypt( input, 0, output, 0);
System.out.println("Plaintext = " + new String( output ) + ".");
```

Sample decryption using a CBC stream

The following code sample illustrates DES decryption in CBC mode:

```java
// Input
```
Developing secure applications

byte[] input = { (byte)0x58, (byte)0x47, (byte)0x50, (byte)0x34, (byte)0x9F, (byte)0xEF, (byte)0x0D, (byte)0x77, (byte)0x31, (byte)0x4E, (byte)0xB7, (byte)0x73, (byte)0x56, (byte)0xAC, (byte)0x3C, (byte)0xD6 }; byte[] iv = { (byte)0xFF, (byte)0xFE, (byte)0xFD, (byte)0xFC, (byte)0xFB, (byte)0xFA, (byte)0xF9, (byte)0xF8 }; byte[] output = new byte[ 16 ]; byte[] keyData = { (byte)0x01, (byte)0x23, (byte)0x45, (byte)0x67, (byte)0x89, (byte)0x01, (byte)0x23, (byte)0x45 }; // Create a new DES key with the given data. DESKey key = new DESKey( keyData ); // Create the DES Engine. DESDecryptorEngine desEngine = new DESDecryptorEngine( key ); // Create the CBC engine. CBCDecryptorEngine cbcEngine = new CBCDecryptorEngine( desEngine, new InitializationVector( iv ) ); // Create a stream from the input byte array. ByteArrayInputStream inputStream = new ByteArrayInputStream( input ); // Now we create the Decryptor using the CBCDecryptorEngine. BlockDecryptor decryptor = new BlockDecryptor( cbcEngine, inputStream ); // Finally, we can read in the decrypted data. int ret = decryptor.read( output, 0, output.length ); decryptor.close(); inputStream.close(); System.out.println("Plaintext = " + new String( output ) + ".");

Block formatters and unformatters

The RIM Cryptographic APIs provides engines that you can use for padding and encoding of blocks of data. These are called BlockFormatterEngines and BlockUnformatterEngines and include the algorithms OAEP, PKCS1, and PKCS5. PKCS5 provides padding, and OAEP and PKCS1 provide block encoding.
A BlockFormatterEngine uses a BlockEncryptorEngine to encrypt the data after the appropriate encoding or padding has been performed. BlockEncryptor uses both a BlockEncryptorEngine and a BlockFormatterEngine, so that you can use either of them with the stream-based method.

The following code sample uses a BlockFormatterEngine:

```java
// Input
byte[] input = { 0x00, 0x00, 0x00, 0x00, 0x00 }; // less than 8 bytes!

// Output
byte[] output = new byte[ 8 ];

// Create a random new DES key.
DESKey key = new DESKey();

// Create engine.
DESEncryptorEngine desEngine = new DESEncryptorEngine( key );

// Create the block encoder engine
PKCS5FormatterEngine formatter = new PKCS5FormatterEngine( desEngine );

// Encode and encrypt the information using PKCS5.
// The last boolean argument indicates whether or not this is the last block.
formatter.formatAndEncrypt( input, 0, input.length, output, 0, true );
System.out.println("Ciphertext = " + new String( output ) + ");
```

The following code sample uses the formatter to encrypt a stream of data with the formatter; this example uses DES in CBC mode with PKCS5 padding.

```java
//Input
byte[] input = { (byte)'H', (byte)'e', (byte)'l', (byte)'l',
(byte)'o', (byte)',', (byte)' ', (byte)'w',(byte)'o', (byte)'r',
(byte)'l', (byte)'d', byte)'!', (byte)'!' }; byte[] iv = { (byte)0xFF, (byte)0xFE, (byte)0xFD,
(byte)0xFC,(byte)0xFB, (byte)0xFA, (byte)0xF9, (byte)0xF8 }; byte[] keyData = { (byte)0x01, (byte)0x23, (byte)0x45,
(byte)0x67,(byte)0x89, (byte)0x01, (byte)0x23, (byte)0x45 }; byte[] keyData = { (byte)0x01, (byte)0x23, (byte)0x45,
(byte)0x67,(byte)0x89, (byte)0x01, (byte)0x23, (byte)0x45 }; byte[] iv = { (byte)0xFF, (byte)0xFE, (byte)0xFD,
```

// Create a new DES key with the given data.
Developing secure applications

```
DESKey key = new DESKey( keyData );

// Create the DES Engine.
DESEncryptorEngine desEngine = new DESEncryptorEngine( key );

// Create the CBC engine.
CBCEncryptorEngine cbcEngine = new CBCEncryptorEngine( desEngine,
new InitializationVector( iv ) );

// Create the PKCS5 Encoder engine.
PKCS5FormatterEngine formatter = new PKCS5FormatterEngine( cbcEngine );

// Create a stream from the input byte array.
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

// Now we create the Encryptor using the CBCEncryptorEngine.
BlockEncryptor encryptor = new BlockEncryptor( formatter, outputStream );

// Finally, we can write the data to encrypt.
encryptor.write( input, 0, input.length );

// We want to close the output stream and grab the bytes.
// NOTE: It is especially important to call close with padding
// encoders, as it ensures that the last block is encoded properly.
encryptor.close();
byte[] output = outputStream.toByteArray();
System.out.println( "Ciphertext = " + new String( output ) + "." );
```

To use a formatter, extend the methods already discussed for encrypting and decrypting data. The following code sample demonstrates the unformatter:

```
// Output from previous example
byte[] output = { (byte)0x58, (byte)0x95, (byte)0xBC, (byte)0xF4,
(byte)0x3F, (byte)0xD8, (byte)0xD8, (byte)0xAD, (byte)0x80,
(byte)0x9A, (byte)0x95, (byte)0x34, (byte)0xB4, (byte)0xCC,
(byte)0x76, (byte)0x79 };
```

//Input
A stream cipher combines plaintext bits with a pseudorandom cipher bit stream (keystream). Plaintext digits are encrypted one at a time.
The stream ciphers portion of the RIM Cryptographic APIs is based upon the common stream interface in Java, and uses similar read and write function calls used to interact with the stream. After a stream cipher is established, all activity is automatically performed by the class on the underlying stream.

This section describes the common uses of stream ciphers in the RIM Cryptographic API and provides sample code for using them. The PseudoRandomSource class and its advantages is discussed, the use of ARC4 (the alleged RC4 algorithm) is demonstrated, and the use of CFB mode for block ciphers is demonstrated.

Pseudo-random sources

A pseudo-random source is a bit stream generator that is initialized with a seed and supplies an arbitrarily long stream of difficult-to-predict and nonrepeating bits.

The PseudoRandomSource class is an abstract class that represents a PRNG. This class provides methods for retrieving bytes from the PRNG and resetting the state of the PRNG.

Stream ciphers

Most stream ciphers create the ciphertext by applying an exclusive-or to the plaintext with a keystream. The keystream is generated from a PRNG.

The ARC4 stream cipher

To use the ARC4 algorithm, an instance of the ARC4PseudoRandomSource class is required. While you can use ARC4PseudoRandomSource directly to encrypt data, the more flexible and reliable way involves using the PRNGEncryptor class, which implements the StreamEncryptor interface.

Encryption

```java
// Input byte array.
byte[] plaintext = new byte[128];

// In this example, we use the Random source to get input bytes. In practice, this is plaintext. This shows how to use RandomSource as well.
RandomSource.getBytes( plaintext );

// Print out the plaintext.
System.out.println("Plaintext = " + new String( plaintext ) + ".");
```
// Now set up the ARC4Key of length 256 bytes.
ARC4Key key = new ARC4Key(256);

// Now create the ARC4PseudoRandomSource
ARC4PseudoRandomSource source = new ARC4PseudoRandomSource(key);

// Create the output stream to store the encrypted data
ByteArrayOutputStream out = new ByteArrayOutputStream();

// Finally, create the PRNGEncryptor stream encryptor, which uses
// the ARC4PseudoRandomSource and an output stream
PRNGEncryptor encryptStream = new PRNGEncryptor(source, out);

// To encrypt, simply call write with the plaintext
encryptStream.write(plaintext, 0, plaintext.length);
encryptStream.close();
byte[] ciphertext = out.toByteArray();

// Print out the ciphertext.
System.out.println("Ciphertext = " + new String(ciphertext) + ".");

Decryption

// Reload the key to get the ciphertext back to plaintext (decrypt).
source = new ARC4PseudoRandomSource(key);

// Create an input stream with ciphertext as the basis
ByteArrayInputStream in = new ByteArrayInputStream(ciphertext);

// Create an instance of an PRNGDecryptor to decrypt the data
PRNGDecryptor decryptStream = new PRNGDecryptor(source, in);

// Decrypt by reading from the stream and storing
// the decrypted data in plaintext
decryptStream.read(plaintext, 0, ciphertext.length);
decryptStream.close();

// Print out the plaintext.
Using CFB mode

CFB mode is a common mode used to convert a block cipher to a stream cipher. The following sample code shows how to encrypt using CFB mode:

```java
// We are going to use DES since it seems to be the most common
// symmetric key algorithm used.
// Set up the input again as random bytes.
byte[] input = new byte[128];
RandomSource.getBytes( input );

// Set up the DESKey.
DESKey key = new DESKey();

// Set up the DESEngine.
DESEncryptorEngine engine1 = new DESEncryptorEngine( key );

// Set up the Initialization vector.
byte[] iv1 = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08 };

// Set up the output stream.
ByteArrayOutputStream outStream = new ByteArrayOutputStream();

// Set up the encryptor.
CFBEncryptor encryptor = new CFBEncryptor( engine1, new InitializationVector( iv1 ), outStream, true );

// Write out the data to be encrypted.
encryptor.write( input, 0, input.length );

// Close the encryptor stream.
encryptor.close();

// Get the information from the underlying data stream.
byte[] output = outStream.toByteArray();
outStream.close();

// Print out the input and the output.
```

System.out.println( "Plaintext = " + new String( plaintext ) + "." );
Factories: a high level approach to encrypting and decrypting data

The RIM Cryptographic API provides two different ways to accomplish encryption and decryption of data: engine classes and factory classes.
The engine class provides a low-level, hands-on way of manipulating the encryption and decryption schemes. While this is effective, it is also complicated and is not the most appropriate method in all cases.

This illustration shows the process by which data is encrypted and decrypted using the EncryptorEngine and DecryptorEngine classes.

![Encryption and Decryption Process](image)

**Figure 10.1** Encrypting and decrypting data using the EncryptorEngine and DecryptorEngine classes

The EncryptorFactory and DecryptorFactory classes allow you to encrypt and decrypt data with relative ease, and can be more suitable if you are unfamiliar with cryptography.

This illustration shows encrypting and decrypting data using the EncryptorFactory and DecryptorFactory classes.

![Encryption and Decryption Process](image)

**Figure 10.2** Encrypting and decrypting data using the EncryptorFactory and DecryptorFactory classes

### Encrypting data using the EncryptorFactory class

The following example illustrates the use of the EncryptorFactory and DecryptorFactory classes to encrypt and decrypt a text string.

The EncryptorFactory class accepts an Initialization Vector or IV. The IV is a byte array of random-seeming data, which used to further randomize generated data. The data in the array is seeded with the data created when the plaintext is encrypted. The following code sample creates and initializes an IV:

```java
// The Initialization Vector used by the EncryptorFactory
```
private static final byte[] IV = {
    (byte)0x21, (byte)0x92, (byte)0x99, (byte)0x2A, (byte)0x27,
    (byte)0x4F, (byte)0xC1, (byte)0xA8
};

// The class

public EncryptorFactoryDemo()
{

    The EncryptorFactory also accepts a string that defines the details of the encryption. The String
    consists of three concatenated substrings that define (in the following order) the encryption type, the
    mode and the formatting type. The following sample code uses the string "DES/CBC/PKCS5", which
    specifies that DES encryption, in CBC mode, with PKCS5 formatting is used to encrypt the data:

    // A String representing the encryption type
    String encryptionType = new String ("DES/CBC/PKCS5");

    // The original message
    String message = new String("This is the test message.");

    // The Output stream - this contains the ciphertext of the message
    ByteArrayOutputStream stream = new ByteArrayOutputStream();

    // The byte array that can accept the ciphertext from the output stream
    byte[] cipherText = new byte[0];

    The final two inputs of the DecryptorFactory are the outputstream used to collect the ciphertext and
    the key used to encrypt the data. The key, in this case a DES key, is created as follows:

    // The key used to encrypt the data
    // This key must also be used to decrypt the data
    DESKey key = new DESKey();

    Note:

    The code samples in this section do not include steps to generate and distribute
    keys. When you implement your own application, you must first choose a protocol,
    and then create and distribute keys.

    The sample code in the section decrypts data using the same instance of a key used
    to encrypt it.
After you have created the arguments, you can create the new EncryptorFactory object and encrypt the plaintext. In the following code sample, the getEncryptorOutputStream method of the EncryptorFactory class is called and the resulting outputstream is stored in the variable encryptor. The encryptor is created within a try clause to catch and display any exceptions that can occur.

```
try {
    //Create the EncryptorFactory
    EncryptorOutputStream encryptor = EncryptorFactory.getEncryptorOutputStream( key.getData(), stream, encryptionType, IV );
    encryptor.write( message.getBytes() );
    encryptor.close();
    cipherText = stream.toByteArray();

    //Output the messages to the console
    //This is included for the sake of the sample
    System.out.println("Original Message: " + message);
    System.out.println("Ciphertext: " + cipherText);
}
```

If the encryption is successful, the ciphertext and the plaintext is outputted to the console.

**Decrypting data using the DecryptorFactory class**

The decryption process is very similar to the encryption process. In this sample, the data is encrypted and decrypted using the same instance of the key.

In the following code sample, the inputstream is created to hold the ciphertext from the encryption process. The ciphertext, like the key, was previously transmitted to the recipient by the sender. The details of the transmission are outside of the scope of this chapter. The getDecryptorInputStream method of the DecryptorFactory class is called and the resulting object is created and stored in a variable of type DecryptorInputStream called decryptor.

Because the data can contain extra padding, a for loop is used to extract the data from the outputstream. The following example uses a loop that reads from the inputstream and exits when it reads the first empty block of data. The decryption routine occurs within a try block.

```
ByteArrayInputStream input = new ByteArrayInputStream( stream.toByteArray() )
DecryptorInputStream decryptor = DecryptorFactory.getDecryptorInputStream( symmetricKey, input );

byte[] decrypted = new byte[0];
byte[] temp = new byte[10];
for( ;; ) {
    int read = decryptor.read( temp, 0, 10 );
    if( read < 0 ) {
        // We are at the end of the stream.
        break;
    }
    // Copy.
    net.rim.vm.Array.resize( decrypted, read + decrypted.length );
    System.arraycopy( temp, 0, decrypted, decrypted.length - read,
        read );
}
if( Arrays.equals( message.getBytes(), decrypted ) ) {
    // Passed.
    System.out.println("Encryption/Decryption Passed.");
} else {
    // Failed
    System.out.println("Encryption/Decryption Failed.");
}

The arrays are compared to ensure that the decryption was successful and a message appears in the
console. If the decryption fails, an exception error messages appears in the console.

In symmetric key cryptography, the same key is used to both encrypt and decrypt the data, so the key
must be carefully protected.

**Key agreement**

Key agreement provides a way for two independent parties to establish a secure channel over an
insecure network. Key agreement protocols generally use both parties’ public and private keys, along
with any algorithm-specific parameters, to create a secret piece of data that, although derived from
public data, is never sent out over the network. In other words, the secret data is created independently
by both parties through the use of the publicly shared information. You can use this secret data as a
secret session key for secure communication.

This section provides an overview of key agreement and common key agreement schemes.
Using key agreement schemes

A common key agreement scheme is the **ECDHKeyAgreement**. It requires the use of a public key and a private key. Each party in a secure communication uses their own private key and the public key of the other party. When each party uses the key agreement algorithm, the same secret data is created.

![Note:]

The sample code in this section assumes the methods `sendPublicKey()` and `receivePublicKey()` exist and enable the exchange of keys between the two clients. If something is sent by one client, it is received by the other, and vice versa.

1. Create the local secret. In the following code sample, the `generateLocalSecret` method is run on the local client to generate the local secret.

   ```java
   public void generateLocalSecret()
   {
   // Create an EC crypto system for key creation
   ECCryptoSystem cryptoSystem = new ECCryptoSystem();
   // Create the first party's public and private keys
   ECKeypair client1KeyPair = new ECKeypair( cryptoSystem );
   }
   ``

2. Send the local (client1) client public key to the remote user. The remote user public key is received using the previously implemented and mutually agreed upon exchange protocol. The local client private key and the remote user public key are then used by the `generateSharedSecret` method to create a local shared secret. The local shared secret is identical to the secret generated by the remote client.

   ```java
   // Send and receive keys
   sendPublicKey( client1KeyPair.getECPublicKey() );
   ECPublicKey client2PublicKey = receivePublicKey();
   // Generate the shared secret for this client.
   byte[] client1Secret =
   ECDHKeyAgreement.generateSharedSecret(
   client1KeyPair.getECPrivateKey(),
   client2PublicKey );
   // Create a shared secret key based on the shared secret
   DESKey secretKey = new DESKey( client1Secret );
   ```
3. Generate the secret key of type DESKey, which is the shared key that is used for communication between clients. In the following code sample, the generateRemoteSecret method creates the remote user shared secret and key:

```java
public void generateRemoteSecret()
{
    // Create an EC crypto system for key creation
    ECCryptoSystem cryptoSystem = new ECCryptoSystem();
    // Create the remote public and private keys
    ECKeyPair client2KeyPair = new ECKeyPair( cryptoSystem );

    // Send and receive the keys
    sendPublicKey( client2KeyPair.getECPublicKey() );
    ECPublicKey client1PublicKey = receivePublicKey();
    // Generate the secret data based upon the client1 public
    // key and client2 private key
    byte[] client2Secret =
        ECDHKeyAgreement.generateSharedSecret(
        client2KeyPair.getECPrivateKey(),
        client1PublicKey );
    // Create a shared secret key based
    // on this secret data
    DESKey secretKey = new DESKey( client2Secret );
}
```

In the preceding examples, both operations occur locally, and the details of the key exchange are not displayed. In your application, the public keys are exchanged over the network, and the private keys must be kept securely by each party. Identical shared secrets are created by each client (client1Secret and client2Secret).
An example of ECMQVkeyagreement

Another, newer key agreement scheme is ECMQVKeyAgreement. This scheme makes use of an additional set of ephemeral keys for added security. Ephemeral keys are keys that are stored and used for only a limited amount of time. This way, if a key is compromised, a new key is created periodically and the compromised key is no longer used.

The code required is very similar to ECDHkeyagreement.

Note:

The following code sample assumes the methods sendPublicKey() and receivePublicKey() exist and enable the exchange of keys between the two clients. If something is sent by one client, it is received by the other, and vice versa.

1. Create the keys. This step is distinct from the previous example in that both a static, or regular keyPair, and an ephemeral, or temporary, keyPair are created.

   public void generateLocalSecret()
   {
   // Create an EC crypto system for key creation
   ECCryptoSystem cryptoSystem = new ECCryptoSystem();
   // Create the first client's static and
   // ephemeral key pairs
   ECKeyPair client1StaticPair =
   new ECKeyPair( cryptoSystem );
   ECKeyPair client1EphemeralPair =
   new ECKeyPair( cryptoSystem );

   2. Send and receive the newly created public and public ephemeral keyPair using a predetermined key exchange protocol.

   // Send and receive public keys
   sendPublicKey( client1StaticPair.getECPublicKey() );
   sendPublicKey( client1EphemeralPair.getECPublicKey() );
   ECPublicKey client2StaticKey = receivePublicKey();
   ECPublicKey client2EphemeralKey = receivePublicKey();

   // Generate the shared secret for this client. This
   // is identical to that generated by client2.
byte[] client1Secret =
ECMQVKeyAgreement.generateSharedSecret(
client1StaticPair.getECPrivateKey(),
client1EphemeralPair,
client2StaticKey,
client2EphemeralKey);

// Create a shared secret key based
// on this secret data
DESKey secretKey = new DESKey( client1Secret );
}

A shared secret is created and a shared secret key is then created using the shared secret. Because the
operation uses ephemeral keys, the generateSharedSecret method uses the clients private key, the
remote user public key, the local ephemeral keyPair, and the remote ephemeral public key.

public void generateRemoteSecret()
{

// Create an EC crypto system for key creation
ECCryptoSystem cryptoSystem = new ECCryptoSystem();

// Create the second client's static and
// ephemeral key pairs
ECKKeyPair client2StaticPair =
    new ECKKeyPair( cryptoSystem );
ECKKeyPair client2EphemeralPair = new ECKKeyPair( cryptoSystem );

// Send and receive the keys
sendPublicKey( client2StaticPair.getECPublicKey() );
sendPublicKey( client2EphemeralPair.getECPublicKey() );
ECPublicKey client1StaticKey = receivePublicKey();
ECPublicKey client1EphemeralKey = receivePublicKey();

// Generate the shared secret for this client.
byte[] client2Secret =
ECMQVKeyAgreement.generateSharedSecret(
client2StaticPair.getECPrivateKey(),
client2EphemeralPair,
client1StaticKey,
client1EphemeralKey);

// Now we can create a shared secret key based
// on this secret data

DESKey secretKey = new DESKey(client2Secret);

}  

This example performs both operations locally and the key exchange is not included. In your applications, the keys are exchanged over the network and the private keys must be kept securely by each party. The shared secrets created by each client (client1Secret and client2Secret) is identical.

This scheme, in effect, uses five keys. The local public and private ephemeral keys, the local private static key, the remote public static key, and the remote public ephemeral key.

The RIM Cryptographic APIs supports the following key agreement schemes:

• Diffie-Hellman Key Agreement: DHKeyAgreement
• Elliptic Curve Diffie-Hellman Key Agreement: ECDHKeyAgreement
• Elliptic Curve Menezes, Qu, and Vanstone Key Agreement: ECMQVKeyAgreement

Key encoders and decoders

Encoded keys have the advantage that they can be transmitted as a complete package (including any required system parameters) so it is clear to the recipient how to decode the message.

Encoding and decoding an RSA private key using PKCS8

Two classes are provided, PublicKeyEncoder and PrivateKeyEncoder, which handle all the encoding and decoding of keys. A common private key encoding scheme is the PKCS8 format. The following code sample demonstrates how to encode and decode RSA private keys using PKCS8:

// Create a new RSA crypto system for key generation
RSACryptoSystem cryptoSystem = new RSACryptoSystem();

// The key pair can hold the RSA keys
RSAKeyPair keyPair = new RSAKeyPair(cryptoSystem);
RSAPrivateKey privateKey = keyPair.getRSAPrivateKey();

// Now encode the key using the PrivateKeyEncoder class
EncodedKey encodedKey = PrivateKeyEncoder.encode( privateKey, "PKCS8" );
byte[] encodedKeyData = encodedKey.getEncodedKey();

The following sample demonstrates how to decode the encoded key:

RSAPrivateKey decodedKey = ( RSAPrivateKey ) PrivateKeyDecoder.decode( encodedKeyData, "PKCS8" );

Encoding and Decoding an EC Public Key using X.509

The following code sample demonstrates how to encode an elliptic curve public key with X.509:

// Create a new EC crypto system for key generation
ECCryptoSystem cryptoSystem = new ECCryptoSystem();

// The key pair can hold the EC keys
ECKeyPair keyPair = new ECKeyPair( cryptoSystem);
ECPublicKey publicKey = keyPair.getECPublicKey();

// Now encode the key using the PublicKeyEncoder class
EncodedKey encodedKey = PublicKeyEncoder.encode( publicKey, "X509" );
byte[] encodedKeyData = encodedKey.getEncodedKey();

The following code sample demonstrates how to decode an elliptic curve public key with X.509:

ECPublicKey decodedKey = ( ECPublicKey ) PublicKeyDecoder.decode( encodedKeyData, "X509" );

**Digests and MACs**

This section describes the digests and MACs found in the API and provides code examples that show how to use them effectively.

**Digests**

Digests are used to convert input data of arbitrary length to a fixed size hash or digest.

The most commonly used hash function in North America is SHA-1, whose digest length is 160 bits, and SHA-2, which has variable digest bit lengths of 256, 384 and 512 bits. These lengths match the security level of the AES candidate (which is to replace DES). The RIM Cryptography API supports SHA-1 (in the
Developing secure applications

class SHA1Digest), and three variants of SHA-2 (in the classes SHA256Digest, SHA384Digest, and SHA512Digest).

The following code sample demonstrates how to use these classes:

```java
// Instantiate any of the different types of SHA
SHA1Digest digest160 = new SHA1Digest();
SHA256Digest digest256 = new SHA256Digest();
SHA384Digest digest384 = new SHA384Digest();
SHA512Digest digest512 = new SHA512Digest();

// Now we can simply use the SHA-1 algorithm to illustrate
// how the rest of the functions work.
byte[] data = new byte[128];
RandomSource.getBytes( data );

// Update the contents of the hash function
// (This is the data that gets hashed)
digest160.update( data );
digest160.update( data, 10, 15 );

// Now get the hash value.
byte[] digestValue = digest160.getDigest();

You can also retrieve the hash value using the following code:

byte[] hashValue = new byte[digest160.getDigestLength()];
digest160.getDigest( hashValue, 0 );
```

The following digests support streaming: DigestInputStream and DigestOutputStream. The following code sample demonstrates how to create an instance of a digest input or output stream, and pass an instance of a digest algorithm into the constructor:

```java
// Create a SHA digest with the default length (160-bit).
SHA1Digest digest = new SHA1Digest();

// Now create a new DigestOutputStream. Passing in a null value
// for the OutputStream parameter means that this stream does
// not pass the data written to it into another output stream; we
// simply want to use the digest functionality.
DigestOutputStream out = new DigestOutputStream( digest, null );
```
// A call to write also updates the digest (assuming data exists
// and contains the message)
    out.write( data );

// Now get the hash value.
    byte[] digestValue = digest.getDigest();

You can use the same method for DigestInputStream, except that the data read from the input stream
(passed into DigestInputStream) is passed through the digest algorithm.

Support for other digests

All of the other digests in the API use the same interface previously discussed. The RIM Cryptographic
API supports the following hash algorithms:

- MD2 Digest
- MD4 Digest
- MD5 Digest
- SHA-1 Digest
- SHA-256 Digest, SHA-384 Digest, SHA-512 Digest
- RIPEMD-128 Digest, RIPEMD-160 Digest

MACs

MACs are essentially keyed hash functions. That is, a key is required for the creation of the hash value
as well as for the verification of that particular hash value. The MAC functionality in the RIM Crypto-
graphic APIs builds upon the digest interface.

The RIM Cryptographic APIs supports the following MACs:

- CBC MAC
- HMAC
- Null MAC

The HMAC class implements the MAC interface and allows for easy use of keyed hash functions. The
code required to use an HMAC is similar to that for digests, except that you require a key to generate the
hash.

// We need to create an HMAC key, which can be an array
// of data of any length. In most cases, a length equal
// to the bit size of the digest is recommended. Random
// data is used here.
    byte[] keyData = new byte[ 20 ];
    RandomSource.getBytes( keyData );
// Create the key
HMACKey key = new HMACKey( keyData);

// Create the SHA digest
SHA1Digest digest = new SHA1Digest();

// Now an HMAC can be created, passing in the key and the
// SHA digest. Any instance of a digest can be used here.
HMAC hMac = new HMAC( key, digest );

// The HMAC can be updated much like a digest
hMac.update( data );
hMac.update( data, 10, 15 );

// Now get the MAC value.
byte[] macValue = hMac.getMAC();

Streaming for HMACs is also supported. Once you have created the HMAC object, you can use a
MACOutputStream.

// Create the MAC output stream, once again passing in null for
// the OutputStream to just use the MAC functionality
MACOutputStream out = new MACOutputStream( hMac, null );

// A call to write also updates the MAC (assuming data exists
// and contains the message)
out.write( data );

// Now get the MAC value.
byte[] macValue = hMac.getMAC();

A MACInputStream class exists, which you can use in a similar fashion to MACOutputStream.

## Signatures

This section describes the most common uses of signatures in the RIM Cryptographic APIs and
provides code examples that demonstrate how to use the signature classes in these cases. Additionally,
this section describes how you can use signatures with both streams and byte arrays.
A signature can include one or more mathematical values, depending on the signing algorithm used; therefore, you need an encoding scheme. A SignatureSigner combines the message to be signed and the signer's private key to yield the mathematical values that make up a signature. A SignatureEncoder then encodes these mathematical values into a byte array for transmission or storage.

Using a DSA signature signer

A signature signer is the interface that is used for all signing in the RIM Cryptographic APIs. The sample code uses the DSA algorithm, which DSA is the most commonly-used signing algorithm. You can apply the method shown in this code sample to RSA and the other algorithms.

1. Create and use a signature signer.

   // Create key pair
   DSACryptoSystem cryptoSystem = new DSACryptoSystem();
   DSAKeyPair keyPair = new DSAKeyPair( cryptoSystem );
   DSAPrivateKey privateKey = keyPair.getDSAPrivateKey();

   // The message to be signed
   String message = new String("Jeans are on sale");

   // Create the signer itself.
   SignatureSigner signer = new DSASignatureSigner( privateKey );
   signer.update( message.getBytes() );

   // Create an X509 signature.
   EncodedSignature signature = SignatureEncoder.encode( signer, "X509" );

   // Get the data of the signature.
   byte[] signatureData = signature.getEncodedSignature();

   // bytes
   String encodingAlgorithm = signature.getEncodingAlgorithm();
   // "X509"

2. Verify the signature.

   // Retrieve public key.
   DSAPublicKey publicKey = keyPair.getDSAPublicKey();
Developing secure applications

// Decode the signature.
DecodedSignature decodedSignature = SignatureDecoder.decode(
    signatureData, "X509" );

// Get signature verifier.
SignatureVerifier verifier = decodedSignature.getVerifier(publicKey);

// Enter message to be verified.
verifier.update( message.getBytes() );

// Verify the signature.
boolean verified = verifier.verify();

// Print out result.
System.out.println("Signature was verified " + verified + ".");

Using signature streams with an ECDSA signature signer

Now that you know how to use the SignatureSigner and the SignatureVerifier, you can use these
components with a signature stream. The signature stream (SignatureSignerOutputStream) allows you
to write data to the stream, and when you close the stream the signature is appended to the stream. The
following code sample demonstrates how to use the signature stream.

// Generate a key pair that we can use for signing (and eventually
for verifying)
ECKeyPair keyPair = new ECKeyPair( new ECCryptoSystem() );

// Get the private key for use with the signature signer.
ECPrivateKey privateKey = keyPair.getECPrivateKey();

// Create the signature signer. We use ECDSA.
ECDSASignatureSigner signer = new ECDSASignatureSigner( privateKey, new MD5Digest() );

// Create the output stream that we can use as the underlying stream.
// For this example we can simply use a byte array output stream.
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
// Create the SignatureSignerOutputStream

    SignatureSignerOutputStream signerStream = new SignatureSignerOutputStream( signer, outputStream );

// Now we want to write out the data to be signed.

    String info = new String("Hello World");

// Sign this information.

    signerStream.write( info.getBytes() );

// Declare the byte arrays for r and s which represent the signature.

    byte[] r = new byte[signer.getRLength()];
    byte[] s = new byte[signer.getSLength()];
    signer.sign( r, 0, s, 0 );

    // Close the output stream (upper layer).

    signerStream.close();

    // Get the stream contents for later.

    byte[] contents = outputStream.toByteArray();

    // Close the underlying output stream.

    outputStream.close();

You can use the SignatureVerifierInputStream to verify the signature that you sent to output in the previous section. The following code sample shows how to verify the signature that was generated using a signature output stream:

    // Get the public key from the key pair we created in the last example.

    ECPublicKey publicKey = keyPair.getECPublicKey();

    // Set up the ECDSASignatureVerifier

    ECDSASignatureVerifier verifier = new ECDSASignatureVerifier( publicKey, new MD5Digest(), r, 0, s, 0 );

    // Set up the input stream (using contents from above).

    ByteArrayInputStream inputStream = new ByteArrayInputStream( contents );

    // Use this to set up the SignatureVerifierInputStream
Developing secure applications

SignatureVerifierInputStream verifierStream = new SignatureVerifierInputStream( verifier, inputStream );

// Read the data from the stream.
byte[] data = new byte[verifierStream.available()];
verifierStream.read( data );

// Use the verifier to check if the signature is correct.
boolean result = verifier.verify();

// Print out the result.
System.out.println("Signature verification was " + result + ".");

Using an RSA signature signer

RSA signatures have the added requirement that messages be encoded by a formatter (see PKCS1SignatureSigner, PSSSignatureSigner). The formatters are used in a similar way to the SignatureSigners. The following code illustrates the use of PSSSignatureSigner and PSSSignatureVerifier:

// The message to be signed
String message = "Hello World.";

// Make the necessary RSA key pair for signing and verifying
RSACryptoSystem cryptoSystem = new RSACryptoSystem();
RSAKeyPair keyPair = new RSAKeyPair( cryptoSystem );

// Create the digest and the salt value
SHA1Digest digest = new SHA1Digest();
byte[] salt = RandomSource.getBytes( digest.getDigestLength() );

// Create the RSASignatureSigner passing in a digest algorithm
// and PSS signature formatter
PSSSignatureSigner signer = new PSSSignatureSigner( keyPair.getRSAPrivateKey(), digest, salt );

// For this example, simply use the signature signer directly,
// even though we could pass it into a stream as before
signer.update( message.getBytes() );

// Finally, we need to encode the signature using X509
EncodedSignature encSignature = SignatureEncoder.encode( signer, "X509" );

The following examples demonstrates how to decode and verify the signature (assuming the encoded signature and RSA key pair exist):

// Decode the encoded signature
DecodedSignature decodedSignature = SignatureDecoder.decode( encSignature.getEncodedSignature(), "X509" );
SignatureVerifier verifier = decodedSignature.getVerifier( keyPair.getPublicKey() );

// Give the verifier the message string
verifier.update( message.getBytes() );

// Now see if it verifies
if( verifier.verify() == true ) {
  System.out.println( "Signature verifies." );
} else {
  System.out.println( "Signature does not verify." );
}

Encoder API

The RIM Encoder API allows you to encode and decode keys/signatures for transport and storage. This section outlines the encoding methods supported by this API and describes the formats that each encoder produces.

Securing the runtime store

BlackBerry smartphones use a registry to store common software information. The RuntimeStore class implements the registry functionality by providing a central location and associated operations to allow applications to share information. To ensure the integrity of specific data, the registry maintains a high level of security by coupling the RuntimeStore object with the ControlledAccess object. You can use the ControlledAccess object to secure your registry data with a key.

For more information, see the RuntimeStore object or the ControlledAccess object.

This section describes the steps involved in adding and retrieving secure data from the RuntimeStore.
To add an item into the RuntimeStore, you must have a key, the item you wish to add, an ID for the item, and a ControlledAccess object. The key is used to protect the item, the ControlledAccess object allows you to secure the item in the registry, and the ID allows you to identify and retrieve the item. In the following example, the key, called ACME is created and added to the RuntimeStore. The key instance is created using the get method of the CodeSigningKey class. The name of the key is supplied to return the specified key.

The item is added to the RuntimeStore using the put method of the RuntimeStore object. The put method accepts the ID of the item (this must be unique), and the ControlledAccess object which itself accepts the item to be placed in the registry and the key that protects it.

The item is now secure within the RuntimeStore and cannot be modified, except by a user who has been signed with the same key. The following code sample demonstrates how to store something in the RuntimeStore:

```java
long MY_DATA_ID = 0x33abf322367f9018L;
Hashtable myHashtable = new Hashtable();
...

// store myHashtable in the RuntimeStore
CodeSignKey codeSigningKey = CodeSigningKey.get( "ACME" ); // use the code signing key with signer id of "ACME"
RuntimeStore.put( MY_DATA_ID, new ControlledAccess( myHashtable, codeSigningKey ); // wrap myHashtable in a ControlledAccess object
```

The following example creates a new RuntimeStore, then uses its get method to return the item associated with the ID. Because the data is protected by a controlledAccess object, the get method accepts the ID associated with the item and the key used to insert the item into the registry, and returns the item.

```java
Hashtable myHashtable = (Hashtable) RuntimeStore.get( MY_DATA_ID, codeSigningKey );
```

Because the code signing key is required for the operation, the item can only be modified by another user who has access to the same key used to insert the item.

The following example retrieves an object from the RuntimeStore without the use of the key.

```java
Hashtable myHashtable = (Hashtable) RuntimeStore.get( MY_DATA_ID );
// Note: no need to unwrap ControlledAccess
```

The object is retrieved from the RuntimeStore if the user matches a key that is used to protect the object in the RuntimeStore. If either of these methods fail due to an incorrect or missing key, a ControlledAccessException is thrown.
1. The RIM Cryptographic API is a collection of classes that allows you to perform which of the following:

A. Provide security for your BlackBerry custom application
B. Encrypt and decrypt data
C. Digitally sign and verify data (secure the integrity of your data)
D. Authenticate data
E. All of the above
F. None of the above

2. Which of the following is correct? Circle all that apply.

A. You must obtain code signatures from RIM before you can load the BlackBerry smartphone applications onto BlackBerry smartphones.
B. You can test BlackBerry smartphone applications that use controlled APIs in the BlackBerry Smartphone Simulator, or on BlackBerry smartphones, even without code signatures.
C. Without signature codes, you cannot load or test applications that use controlled APIs.
D. You can test BlackBerry smartphone applications that use controlled APIs in the BlackBerry Smartphone Simulator, even without code signatures.

3. Key cryptography is either symmetric, or asymmetric. Which of the following statements correctly describes the difference between the two forms of key cryptography?

A. In symmetric key cryptography, two keys are used which are identical in size, while in asymmetric key cryptography, the two keys can be different sizes.
B. In symmetric key cryptography, a single key is used for both encrypting and decrypting, while in asymmetric key cryptography, a pair of keys is used; one key is used to encrypt a file, and the other is used to decrypt it.
C. In symmetric key cryptography, all users in a system share a common key, while in asymmetric key cryptography, a common key is used to encrypt files, but each device has a unique key that is used to decrypt files.
D. In both symmetric and asymmetric key cryptography, a public/private key pair is used. In symmetric key cryptography, the two keys have the same bit length, while in asymmetric key cryptography, the public key is larger than the private key.

4. When a public/private key pair is used, which of the following is true:

   A. The content of both keys must be kept secret.
   
   B. The content of both keys can be shared.
   
   C. The public key that can be shared, while the private key contains secret information, and must be protected.
   
   D. The private key that can be shared, while the public key contains secret information, and must be protected.

5. Which of the following statements is true?

   A. A certificate is an object that binds the identity of an entity to a public key.
   
   B. A certificate is an object that binds a private key to a public key.
   
   C. A certificate is an object that binds a public/private key pair to a BlackBerry smartphone.
Answers

1. E
2. A, D
3. C
4. C
5. A
Supported authentication models

BlackBerry Enterprise Server version 5.0 supports the following authentication models:

- BlackBerry Administration Service authentication (default)
- mailbox authentication (IBM® Lotus® Domino® environment only)
- Microsoft® Active Directory® authentication

BlackBerry Enterprise Server version 5.0 does not support Windows® authentication or Microsoft® SQL Server® authentication.

You must create one of the following administrator accounts with sufficient credentials to authenticate with the BlackBerry Administration Service:

- administrator in the BlackBerry Administration Service
- mailbox administrator (Lotus Domino environment only)
- administrator in Microsoft Active Directory

Roles or capabilities that are based on database user accounts are not supported in BlackBerry Enterprise Server version 5.0.

For more information about how to create a BlackBerry Administration Service administrator, visit www.blackberry.com/go/serverdocs to see BlackBerry Enterprise Server Administration Guide.

Authentication credentials

In the BlackBerry Enterprise Server User Administration Tool version 5.0, you can use the following parameters interchangeably:

- -username or -sqluser (authentication user name)
- -password or -sqlpass (authentication password)

The BlackBerry Enterprise Server User Administration Tool uses the values associated with these parameters for the authentication and authorization models that are supported in the BlackBerry® Enterprise Server version 5.0. The BlackBerry Enterprise Server User Administration Tool does not support Microsoft SQL Server authentication or roles or capabilities that are defined on the Microsoft SQL Server database engine.
Syntax for authentication credentials

In the BlackBerry Enterprise Server User Administration Tool, the variable `<credentials>` represents the user name and password that you use for authentication with the BlackBerry Administration Service.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-username &lt;user name&gt;</td>
<td>authentication user name</td>
</tr>
<tr>
<td>-sqluser &lt;user name&gt;</td>
<td></td>
</tr>
<tr>
<td>-password &lt;password&gt;</td>
<td>authentication password</td>
</tr>
<tr>
<td>-sqlpass &lt;password&gt;</td>
<td></td>
</tr>
<tr>
<td>-domain &lt;domain&gt;</td>
<td>authentication domain</td>
</tr>
<tr>
<td>-bas_auth</td>
<td>use BlackBerry Administration Service authentication (default)</td>
</tr>
<tr>
<td>-mailbox_auth</td>
<td>use mailbox authentication (Lotus Domino environment only)</td>
</tr>
<tr>
<td>-ad_auth</td>
<td>use Microsoft Active Directory authentication</td>
</tr>
</tbody>
</table>

Configuring authentication credentials

The BlackBerry Enterprise Server User Administration Tool uses the `-set_client_auth <credentials>` `-set_p <password>` command to store credentials in the Windows registry, and then insert the credentials into the command line options.

```
BESUserAdminClient -username admin -password password -
set_client_auth "-username smoser -password password1 -ad_auth -
domain test.rim.net" -set_p password
```

This creates the following Windows registry entry:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Research In Motion\BlackBerry Resource Kit\BESUserAdmin\ClientAuth
```
Quiz

1. Which of the following is true?
   
   A. Roles or capabilities that are based on database user accounts are not supported in BlackBerry Enterprise Server version 5.0.
   
   B. Roles or capabilities that are based on database user accounts are fully supported in BlackBerry Enterprise Server version 5.0.
   
   C. Roles or capabilities that are based on database user accounts are supported in BlackBerry Enterprise Server version 5.0, but only if your administrator enables this option.

2. BlackBerry Enterprise Server version 5.0 supports which of the following authentication models?
   
   A. BlackBerry Administration Service authentication (default)
   
   B. Mailbox authentication (IBM® Lotus® Domino® environment only)
   
   C. Microsoft® Active Directory® authentication
   
   D. All of the above
   
   E. None of the above.
Chapter 9

Answers

1. A

2. D
Using encryption in an application

Data encryption on the BlackBerry smartphone

Administrators can create an IT policy to require that all BlackBerry smartphone user data stored in the BlackBerry smartphone applications is encrypted locally in flash memory. You can create a BlackBerry Java Application that uses APIs to register the data so that the encryption service encrypts the data with the same security key before storing it in flash memory.

Data encryption in transport

If you use the BlackBerry Enterprise Server as the network gateway for your application, the BlackBerry Enterprise Server encrypts data using AES or TripleDES encryption at all points in the connection between the BlackBerry smartphone and the BlackBerry Enterprise Server behind the organization’s firewall. If you require data to be encrypted further between the BlackBerry Enterprise Server and the destination server, you can use the HTTPS protocol and use SSL/TLS encryption.

If your application uses the BlackBerry® Internet Service or the Internet gateway of the wireless service provider, data traffic is not encrypted. If your BlackBerry smartphone users prefer, you can use HTTPS to encrypt the data, or you can use the Java® APIs for encryption to apply your own symmetric key or public key cryptography.

Interface PublicKey

All superinterfaces:

    Key, Persistable, net.rim.vm.Persistent

All known implementing classes:

    RSAPublicKey, KEAPublicKey, ECPublicKey, DSAPublicKey, DHPublicKey

public interface: PublicKey

extends Key

PublicKey is an interface which represents the idea of a public key.

Category

Signed: Only signed clients can access this element. If you intend to use this element, contact RIM to establish the necessary agreements that allows you to have your COD files signed. Signing is required for
your code to run on a BlackBerry smartphone; however, you can carry out development under the BlackBerry Java Plug-in for Eclipse without signing the Code.

<table>
<thead>
<tr>
<th>Category</th>
<th>CryptoSystem</th>
<th>getCryptoSystem—Returns the CryptoSystem used by this public key.</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed</td>
<td>void</td>
<td>verify—Performs certain integrity checks on the public key parameters.</td>
</tr>
</tbody>
</table>

**Method detail**

getCryptoSystem

```java
public CryptoSystem getCryptoSystem()
Returns the CryptoSystem used by this public key.
Returns:
The crypto system.
```

**Category**

Signed: Only signed clients can assess this element. If you intend to use this element, contact RIM to establish the necessary agreements that allows you to have your COD files signed. Signing is required for your code to run on a BlackBerry smartphone; however, you can carry out development under the BlackBerry Java Plug-in for Eclipse without signing the CODs.

**Verify**

```java
public void verify()
throws InvalidKeyException,
InvalidCryptoSystemException,
CryptoTokenException,
CryptoUnsupportedOperationException
```

Performs certain integrity checks on the public key parameters. These checks are useful to prevent certain types of attacks that involve modifying the parameters, and then using a signed message to calculate the private key parameters.

**Throws**

- `InvalidKeyException`—Thrown if the specified key is invalid or malformed.
- `InvalidCryptoSystemException`—Thrown if the specified crypto system is invalid.
- `CryptoTokenException`—Thrown if an error occurs with the crypto token or the crypto token is invalid.
- `CryptoUnsupportedOperationException`—Thrown if a call is made to an unsupported operation.

**Category**
Signed: Only signed clients can access this element. If you intend to use this element, contact RIM to establish the necessary agreements that allow you to have your COD files signed. Signing is required for your code to run on a BlackBerry smartphone; however, you can carry out development under the BlackBerry Java Plug-in for Eclipse without signing the CODs.
1. Which of the following is true?

A. If your application uses the BlackBerry Internet Service or the Internet gateway of the wireless service provider, data traffic is automatically encrypted.

B. If your application uses the BlackBerry Internet Service or the Internet gateway of the wireless service provider, data traffic is not encrypted.

C. If your application uses the BlackBerry Internet Service, data traffic is automatically encrypted. If your application uses the Internet gateway of the wireless service provider, data traffic is not encrypted.

D. If your application uses the BlackBerry Internet Service, data traffic is not encrypted. If your application uses the Internet gateway of the wireless service provider, data traffic is automatically encrypted.
Answers

1. B
BlackBerry Signing Authority Tool

The BlackBerry Signing Authority Tool is server-side software application that you can use to manage access to specified APIs and data stores.

The BlackBerry Signing Authority Tool uses asymmetric private/public key cryptography to validate the authenticity of a signature request, thereby protecting the data and intellectual property of your applications. You can configure it to restrict access to specific APIs and data stores by confining the signing of applications to internal developers.

You can also configure the BlackBerry Signing Authority Tool to allow external developers to request and receive signatures for accessing specified APIs and data. You can track (and accept or reject) signature requests based on administrator control, thus the BlackBerry Signing Authority Tool can assist in the monitoring and enforcement systems for license agreements as they relate to APIs and application data that requires signing by the administrator.

The BlackBerry Signing Authority Tool supports all versions of the BlackBerry Java Plug-in for Eclipse and applications created for Java-based BlackBerry smartphones. The code signing process permits you to use a controlled package, class, or method in a BlackBerry smartphone application. The BlackBerry Signing Authority tool enables the administrator to request, receive, and verify permissions (code signatures) in a secure environment.

<table>
<thead>
<tr>
<th>Benefit of authorizing code files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API access control</td>
<td>protect and control access to sensitive, internal APIs</td>
</tr>
<tr>
<td>Data access control</td>
<td>protect and control access to persistent data in the runtime store</td>
</tr>
<tr>
<td>License enforcement</td>
<td>enforce and track licence agreements</td>
</tr>
</tbody>
</table>

Public-key cryptography

The BlackBerry Signing Authority Tool uses public-key cryptography to authorize and authenticate code. Public-key cryptography uses a pair of keys to sign and verify messages. The public key is distributed to clients, while the private key is kept confidential and is used only by the originating server. Messages signed with the private key can be verified only with the public key. Both keys are required for the secure authenticated transfer of messages.
BlackBerry Signing Authority Tool components

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlackBerry Signing Authority Tool</td>
<td>Create the initial private/public key pair, distribute the public key, and manage key security.</td>
</tr>
<tr>
<td>Remote Client Administration</td>
<td>Create a database for clients that are allowed to request code signing. If you want to request permission to use code, you must be registered in a database that is managed by this application.</td>
</tr>
<tr>
<td>File Signer</td>
<td>Manage communications between developers and Web Administration databases. The File Signer receives, evaluates, and grants permissions for code signing requests.</td>
</tr>
<tr>
<td>Proxy servlet</td>
<td>Use with a proxy servlet in the DMZ to secure File Signer and BlackBerry Signing Authority Tool information. The proxy servlet enables requests to reach the File Signer without security breaches.</td>
</tr>
</tbody>
</table>

**Note:**
The File Signer was developed and tested using the JRun Proxy servlet, which is the recommended proxy. However, you can install and configure other proxy servlets. Always install the File Signer application behind a firewall.

**Security overview**

The BlackBerry Signing Authority Tool uses public-private key cryptography to authorize and authenticate code.

- The private key is kept confidential using a password. This key is used only by the code signing system.
- The public key is distributed and appended to every API file that requires protection.

Messages signed with the private key can be verified by the correct public key. If code is signed by any key other than the correct half of a key pair, the code cannot run correctly.

**Managing private key passwords**

Security in the BlackBerry Signing Authority Tool depends on the private key password. The password must be exactly eight characters in length, and contain uppercase and lowercase characters, digits, and punctuation characters. The password must be hard to guess and must not appear in a dictionary. Never write the password down.
Configuring the BlackBerry Signing Authority Tool

To configure the BlackBerry Signing Authority Tool, perform the following actions:

1. Verify the quality of data.
2. Create a public-private key pair.
3. Distribute the public key.
4. Create a back-up copy of the private key.
5. Verify the quality of data.

When you open the BlackBerry Signing Authority Tool for the first time, use the RIM RNG Service to verify the quality of the random data.

1. Open the RNG Monitor.
2. Click Details.
3. Observe the color traffic-light graphic at the top of the BlackBerry Signing Authority Tool window.
1. Which of the following is true?
   A. The BlackBerry Signing Authority Tool is a web-based application that you can use to protect your applications from unauthorized duplication and download.
   B. The BlackBerry Signing Authority Tool is server-side software application that you can use to manage certificates on BlackBerry smartphones.
   C. The BlackBerry Signing Authority Tool is server-side software application that you can use to manage access to specified APIs and data stores.

2. Which of the following is true?
   A. The BlackBerry Signing Authority Tool uses symmetric key cryptography to authorize and authenticate code.
   B. The BlackBerry Signing Authority Tool uses public-private key cryptography to authorize and authenticate code.
   C. The BlackBerry Signing Authority Tool does not use key cryptography.
Answers

1. C
2. B
Developing secure applications

Many applications on BlackBerry smartphones require some degree of security. You can customize the level of security to suit your individual application. Using cryptography, you can encrypt communication, to meet the following four objectives:

- **Confidentiality**—ensure that information can be understood only by the intended recipient.
- **Integrity**—ensure that information cannot be altered in storage or transit between sender and intended recipient without the alteration being detected.
- **Nonrepudiation**—ensure that the creator/sender of the information cannot later deny his or her intentions in the creation or transmission of the information.
- **Authentication**—ensure that the sender and recipient can each confirm the other’s identity, and that each can confirm the origin/destination of the information.

The RIM Cryptographic API provides tools you can use to complete the following tasks:

- Encrypt and decrypt data
- Digitally sign and verify data (secure the integrity of your data)
- Authenticate data

Summary
Review Questions

1. List and describe the three primary goals of cryptography.

2. Discuss the differences between symmetric keys, and asymmetric key pairs, and describe the relative merits of each.

3. Explain the difference between block ciphers and stream ciphers.

4. Explain the difference between engine classes and factory classes.

5. Describe the purpose of a certificate.

6. Describe the purpose of a keystore.