SYBASE®

# Using Cryptography in PowerBuilder 10.0

PowerBuilder Engineering, Information Technology and Solutions Group

technology

WHITE PAPER

## Table of Contents

## Overview

PowerBuilder® continues to excel as a feature-rich development environment for building data-aware applications. The venerable DataWindow®, which has expanded its capabilities in every release of the product, now includes rich XML generation and manipulation routines.

Internet development is a risky proposition; around the corner from every router, there could be someone tapping into the wire and intercepting TCP/IP packets in an attempt to gain access to someone else's data. Security is an increasingly important aspect of application development, one that developers cannot ignore. Prior to version 10, the PowerBuilder platform lacked a strongly *integrated* cryptography solution. A few articles were available that demonstrated how to invoke calls in external DLLs, but no part of the product itself truly allowed the PowerBuilder developer to invoke cryptographic operations without leaving PowerScript®.

PowerBuilder 10 introduces a new cryptographic object, the `PBCrypto` proxy object. This proxy object exposes a wide array of cryptographic operations suitable for use inside PowerBuilder objects. The PowerBuilder developer now has easy access to the entire gamut of cryptographic operations, from digital signatures to message digests and powerful symmetric encryption algorithms. The purpose of this paper is to educate the developer on what algorithms are available, explain some of the design decisions behind the architecture, and show what extension points exist for this exciting new PowerBuilder functionality.

This paper *does not* aim to explain the proper use of cryptographic functions or the way they work. A coupon included with each license of PowerBuilder 10 offers a discount toward the purchase of a book that covers these subjects: *Java Cryptography Extensions: Practical Guide for Programmers* by Jason Weiss (ISBN: 0-12-742751-1).

The `PBCrypto` proxy object is actually a veneer laid over the Java Cryptography Extensions (JCE) that builds on the growing Java integration offered by the PowerBuilder IDE. There are many benefits to this architecture, especially the ability it gives developers to plug in cryptographic implementations from a wide array of vendors and make use of a vast amount of documentation on Java's cryptography platform.

**It is important to realize that although the `PBCrypto` proxy object included in PowerBuilder 10 has its roots in Java and the JCE, it is possible for the PowerBuilder developer to use the cryptographic algorithms without having to learn Java or the JCE.**

## Architecture of the PBCrypto Non-Visual Object

Cryptography operations rarely result in output that consists entirely of printable characters. One of the first obstacles in performing cryptography with PowerBuilder is that the PowerScript language does not offer a true `byte` data type. To overcome this problem, nearly all of the cryptographic operations declared in the `PBCrypto` proxy rely on Base64-encoded arguments and return Base64-encoded results. Base64 encoding is defined in an RFC and effectively turns 8-bit binary data into 6-bit printable characters.

The `PBCrypto` proxy and the `n_cst_cryptography` non-visual object that knows how to create an instance of the proxy object reside in the library named **pbcryptoclient100.pbd**, which must be in the PowerBuilder library list in addition to the **pbejbclient100.pbd**. Both libraries should be located in the .\Program Files\Sybase\Shared\PowerBuilder directory. (The **pbcryptoclient100.pbd** is installed in Program Files\Sybase\PowerBuilder 10.0\Cryptograph.**)** Additionally, the solution relies on three Java archive (.jar) files that must be present in the Java CLASSPATH and accessible to PowerBuilder. The three libraries are:

- Java Cryptography Extensions library found in jce.jar, obtainable from the JDK installation (JDK 1.4 or later)
- A JCE provider, such one available from Bouncy Castle at http://bouncycastle.org, bcprov-jdk14-1xx.jar, where xx represents the version number
- The Java/PowerBuilder veneer library, pbcrypto-1_0.jar

The last library, pbcrypto-1_0.jar, is the entry point into the Java environment from the PowerBuilder environment. In fact, the `PBCrypto` proxy was generated from one of the Java classes inside the pbcrypto-1_0.jar file.

### Mapping JCE Exceptions into PowerBuilder Exceptions

To keep the design simple, all of the JCE exceptions have been reduced to a single PowerBuilder exception that the developer must catch. That exception is the `NoSuchAlgorithmException`. The inherent plug-in architecture of the JCE makes it possible to request a cryptographic algorithm by name, and if the installed provider's base does not offer an algorithm with that name, the `NoSuchAlgorithmException` is thrown.

Although it would have been possible to cascade each of the JCE exceptions into the PowerBuilder environment, that would have required developers to learn more about the intrinsic JCE operations. Future versions of this cryptographic library might cascade more exceptions back to the PowerBuilder developer, depending on developer feedback to product management.

### Obtaining an Instance of the PBCrypto Proxy Object

The `PBCrypto` proxy is accessible only through the `n_cst_cryptography` non-visual object. The `n_cst_cryptography` NVO declares a single method modeled after a Singleton design pattern (though the PowerBuilder environment makes no guarantee that there will only be a single instance of the NVO; the NVO is marked with AutoInstantiate set to true). Whenever the developer needs to call a cryptographic operation, the `getInstance()` method is invoked on `n_cst_cryptography`, like this:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto

TRY
      IF lnv_crypto.of_getInstance(pbcrypto) THEN
            //
            // Do your crypto calls here
            //
      END IF
CATCH (NoSuchAlgorithmException nsae)
      MessageBox("Alert", nsae.getMessage())
END TRY
```

Developers are encouraged to use the preceding code example as a template in their applications. One of the primary objectives in engineering a PowerBuilder cryptography solution was simplicity for the PowerBuilder developer. As demonstrated in this code example, within a single `TRY…CATCH` block with a single method invocation, it is possible to obtain an instance of a `PBCrypto` proxy object and begin invoking cryptographic operations.

## PBCrypto Proxy API

The `PBCrypto` proxy wraps 13 different cryptographic methods declared inside of the pbcrypto-1_0.jar library. This section discusses each method in detail, points to the window inside the demo application that shows the method being invoked, and references the chapter or section of the book *Java Cryptography Extensions: Practical Guide for Programmers* where developers can learn more about the algorithm and its use. The methods are presented alphabetically, as documented by the PowerBuilder Object Browser.

### CreateRSAKeyPair

creatersakeypair ( ) returns any
throws nosuchalgorithmexception

RSA Key Pairs are typically used in digital signature operations or asymmetric encryption operations where the amount of plaintext is extremely small—for example, encrypting a secret key. Asymmetric encryption is discussed throughout Chapter 3, and digital signatures are discussed in Chapter 4, Sections 4.4 and 4.5.

The sample code provided with the `PBCrypto` solution demonstrating this method can be found in the `w_digital_signature.cb_create.clicked()` method. The following example uses this code:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto
String ls_keys[2]

TRY
      IF lnv_crypto.of_getInstance(pbcrypto) THEN

            ls_keys = pbcrypto.createRsaKeyPair()
            mle_public.text = ls_keys[1]
            mle_private.text = ls_keys[2]

      END IF
CATCH (NoSuchAlgorithmException nsae)
      MessageBox("Alert", nsae.getMessage())
END TRY
```

It is important to remember that cryptography operations rarely result in printable characters, and that the result from the `createRsaKeyPair()` method is automatically Base64 encoded. Creating RSA key pairs is inherently time consuming, and depending upon the speed of your machine, the generation could take as much as 15 seconds. Because of this slow generation speed, it is advisable to display some type of feedback to the user.

**Note:** The public key is always the first entry in the array, and the private key is always the second entry in the array.

### DecryptCipherTextUsingBlockCipher

decryptciphertextusingblockcipher ( string algorithm, string secretKey, string cipherText ) returns string
throws nosuchalgorithmexception

Symmetric block ciphers include encryption algorithms like the Advanced Encryption Standard (AES), the replacement for the aging DES from NIST. Symmetric ciphers are discussed throughout Chapter 2. The first argument is the name of the cipher algorithm that should be used—for example, “AES” or “DES” or “Blowfish” or any other cipher algorithm installed and available to the JCE. The second argument is the Base64-encoded secret key used in the original encryption operation, and the third argument is the cipher text, Base64 encoded.

For complete details on what algorithms are available for use in the first argument, review Chapter 1, Sections 1.4 through 1.6.

The sample code provided with the `PBCrypto` solution demonstrating this method can be found in the `w_symmetric_encryption.cb_decrypt.clicked()` method. The following example uses this code:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto

TRY
      IF lnv_crypto.of_getInstance(pbcrypto) AND
            (Len(sle_encrypted.text) > 0) THEN
      sle_decrypted.text =
            pbcrypto.decryptCipherTextUsingBlockCipher("AES",
                  sle_secretKey.text,
                  sle_encrypted.text)
      ELSE
            MessageBox("Alert", "You must encrypt a phrase first!")
      END IF
CATCH (NoSuchAlgorithmException nsae)
      MessageBox("Alert", nsae.getMessage())
END TRY
```

Both the secret key and the cipher text (encrypted text) are expected to be Base64 encoded when they are passed. Passing unencoded values will result in a bogus decryption. Plain text is returned from the method, and it is not Base64 encoded.

It is important to check that the arguments to the method are not null and do not contain the empty string.

### DecryptCipherTextUsingPBE

decryptciphertextusingpbe ( string algorithm, long salt[], long iters, character passphrase[], string cipherText ) returns string throws nosuchalgorithmexception

PBE stands for Password Based Encryption. Unlike other symmetric block ciphers where a secret key is system generated using a cryptographically secure pseudorandom number generator, this method relies on having someone enter a passphrase/password. The inherent risk here is that the password might not a strong combination of letters, numbers, and non-dictionary words.

PBE is discussed in Chapter 2, Section 2.8. The first argument is the name of the algorithm, the second is a random set of salt values to prime the cipher algorithm, the third is the number of iterations to perform while priming the cipher algorithm, the fourth is an array of characters representing the passphrase, and the fifth is the Base64-encoded ciphertext. Plain text is returned from the method, and it is not Base64 encoded.

For complete details on what algorithms are available for use in the first argument, review Chapter 1, Sections 1.4 through 1.6.

The sample code provided with the `PBCrypto` solution demonstrating this method can be found in the `w_pbe_encryption.cb_decrypt.clicked()` method. The following example uses this code:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto

//The salt and iterations must be the same used in the encryption.
//Salt and iterations are not considered sensitive and can be
//shared using open, unsecured channels.
```

```
long salt[] = { 23, 197, 85, 82, 13, 56, 213, 197 }
long iterations = 1000

TRY
      IF lnv_crypto.of_getInstance(pbcrypto) AND
            (Len(sle_passphrase.text) > 0) AND
            (Len(sle_plaintext.text) > 0) THEN
            sle_decrypted.text =
                  pbcrypto.decryptCipherTextUsingPBE("PBEWithMD5AndDES,
                        salt,
                        iterations,
                        sle_passphrase.text,
                        sle_encrypted.text)
      ELSE
            MessageBox("Alert", "Enter a passphrase/message first!")
      END IF
CATCH (NoSuchAlgorithmException nsae)
      MessageBox("Alert", nsae.getMessage())
END TRY
```

It is important to check that the arguments to the method are not null and do not contain the empty string.

## DecryptSecretKeyUsingRSAPrivateKey

decryptsecretkeyusingrsaprivatekey ( string string_1, string string_2 ) returns string
throws nosuchalgorithmexception

The RSA asymmetric algorithms support encryption, albeit of a finite and relatively small amount of data. The combination of symmetric and asymmetric encryption is demonstrated in Chapter 5, Section 5.4.6. The technique is powerful because symmetric ciphers are much faster and can handle more data than asymmetric ciphers. However, securely distributing the secret key of a symmetric cipher is a challenge. The architecture in this example demonstrates how to use the best features of both types of ciphers.

The secret key is used to encrypt the large plain text block. Then the secret key is encrypted using the public RSA key of the intended recipient. The cipher text and the encrypted secret key can then be distributed over insecure channels, where the recipient uses a private key to decrypt the secret key, then uses the secret key to decrypt the cipher text, revealing the original plain text of the message. The first argument is the Base64-encoded secret key that was encrypted using the RSA public key. The second argument is the Base64-encoded representation of the RSA private key. The result of the method is a Base64-encoded representation of the secret key.

The sample code provided with the `PBCrypto` solution demonstrating this method can be found in the `w_symmetric_asymmetric_combo.cb_decrypt.clicked()` method. The following example uses this code:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto

TRY
      IF lnv_crypto.of_getInstance(pbcrypto) THEN
            sle_decrypted.text =
      pbcrypto.decryptSecretKeyUsingRsaPrivateKey(sle_encrypted.text,
            mle_private.text)
      END IF
CATCH (NoSuchAlgorithmException nsae)
      MessageBox("Alert", nsae.getMessage())
END TRY
```

It is important to check that the arguments to the method are not null and do not contain the empty string. This particular demo window provides specific controls that ensure the button cannot be clicked unless values are present.

### DumpBCProviderList

dumpbcproviderlist ( string engine )  returns string

This is merely a utility method to help PowerBuilder developers search for and identify which algorithms are available for a particular cryptographic engine. They can use it, for example, to check which message digest algorithms or which symmetric cipher algorithms are available. For complete details on what algorithms are available, review Chapter 1, Sections 1.4 through 1.6. The sole argument of the method is the formal name of the cryptographic engine to inspect.

The sample code provided with the `PBCrypto` solution demonstrating this method can be found in the `w_dump_available_algorithms.ddlb_engine.selectionchanged()` method. The following example uses this code:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto

IF lnv_crypto.of_getInstance(pbcrypto) THEN
      mle_dump.text = pbcrypto.dumpBCProviderList(this.text)
END IF
```

Engine names are well-defined JCE entities. For additional coverage of what engines are available beyond those documented in the JCE book, see the Java Cryptography Extension document located at http://java.sun.com.

### EncryptPlainTextUsingBlockCipher

encryptplaintextusingblockcipher ( string algorithm, string secretKey, string plainText )  returns string
throws nosuchalgorithmexception

This method is directly related to the decryptCipherTextUsingBlockCipher() method covered earlier in this paper. For chapter and sample information, please review that method. The first argument is the algorithm name, the second is the Base64-encoded secret key, and the third is the plain text to be encrypted. Cipher text is returned from the method, Base64 encoded.

The following example shows how to invoke this method:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto

TRY
      IF lnv_crypto.of_getInstance(pbcrypto) AND
            (Len(sle_secretKey.text) > 0) THEN
            sle_encrypted.text =
                  pbcrypto.encryptPlainTextUsingBlockCipher("AES",
                        sle_secretKey.text,
                        sle_plaintext.text)
      ELSE
            MessageBox("Alert", "You must generate a secret key first!")
      END IF
CATCH (NoSuchAlgorithmException nsae)
      MessageBox("Alert", nsae.getMessage())
END TRY
```

## EncryptPlainTextUsingPBE

encryptplaintextusingpbe ( string algorithm, long salt[], long iters, character passphrase[], string plainText ) returns string
throws nosuchalgorithmexception

This method is directly related to the decryptCipherTextUsingPBE() method covered earlier in this paper. For chapter and sample information, please review that method. The first argument is the name of the algorithm, the second is a random set of salt values to prime the cipher algorithm, the third is the number of iterations to perform while priming the cipher algorithm, the fourth is an array of characters representing the passphrase, and the fifth is the plain text to encrypt. Cipher text is returned from the method, Base64 encoded.

The following example shows how to invoke this method:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto

long salt[] = { 23, 197, 85, 82, 13, 56, 213, 197 }
long iterations = 1000
//char password[] = { 'a', 'p', 'p', 'l', 'e', 'j', 'a', 'x'}

TRY
      IF lnv_crypto.of_getInstance(pbcrypto) AND
            (Len(sle_passphrase.text) > 0) AND
            (Len(sle_plaintext.text) > 0) THEN
            sle_encrypted.text =
                  pbcrypto.encryptPlainTextUsingPBE("PBEWithMD5AndDES",
                        salt,
                        iterations,
                        sle_passphrase.text,
                        sle_plaintext.text)
      ELSE
            MessageBox("Demo Alert", "Enter passphrase/message first!")
      END IF
CATCH (NoSuchAlgorithmException nsae)
      MessageBox("Alert", nsae.getMessage())
END TRY
```

## EncryptSecretKeyUsingRSAPublicKey

encryptsecretkeyusingrsapublickey ( string string_1, string string_2 ) returns string
throws nosuchalgorithmexception

This method is directly related to the decryptSecretKeyUsingRSAPrivateKey() method covered earlier in this paper. For chapter and sample information, please review that method. The first argument is the Base64-encoded representation of the secret key, the second is the RSA public key, also Base64 encoded. The result of the method is a Base64-encoded representation of the encrypted secret key.

The following example shows how to invoke this method:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto

TRY
      IF lnv_crypto.of_getInstance(pbcrypto) THEN
            sle_encrypted.text =
pbcrypto.encryptSecretKeyUsingRsaPublicKey(sle_secretkey.text,
mle_public.text)
```

```
      END IF
CATCH (NoSuchAlgorithmException nsae)
      MessageBox("Alert", nsae.getMessage())
END TRY
```

### GenerateDigitalSignature

generatedigitalsignature ( string keyAlgorithm, string signatureAlgorithm, string privateKey, string document ) returns string

Digital signatures are a powerful mechanism relying on the properties of asymmetric encryption and message digesting to ensure that the text or body of a document has not been manipulated or changed in any way during transmission from one person to another. One of the more popular misconceptions is that a digital signature manipulates a document. In fact, a digital signature can be simplified and thought of as a checksum of a document sent as a separate attachment to the original document. Thus, the original document is never modified in any way when a digital signature is generated.

Asymmetric encryption is discussed throughout Chapter 3, and digital signatures are discussed in Chapter 4, Sections 4.4 and 4.5. The first and second arguments contain the key and signature algorithms, respectively. The third argument is the Base64-encoded representation of the RSA private key, and the fourth argument is the plain text body of the document to be signed. The result is a Base64-encoded representation of the digital signature for the document.

The sample code provided with the `PBCrypto` solution demonstrating this method can be found in the `w_digital_signature.cb_sign.clicked()` method. The following example uses this code:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto

TRY
      IF lnv_crypto.of_getInstance(pbcrypto) THEN
            sle_signature.text =
                  pbcrypto.generateDigitalSignature("RSA",
                        "MD5withRSA",
                        mle_private.text,
                        mle_message.text)
      END IF
CATCH (NoSuchAlgorithmException nsae)
      MessageBox("Alert", nsae.getMessage())
END TRY
```

For complete details on what algorithms are available for use in the first and second arguments, review Chapter 1, Sections 1.4 through 1.6.

### GenerateMessageAuthenticationCode

generatemessageauthenticationcode ( string algorithm, string secretKey, string document ) returns string

Message Authentication Codes (MAC) take the concept of a message digest one step further. During transmission of a document and a message digest between two people, Alice and Bob, it is possible for a third, Eve, to intercept both items, modify the document, and then modify the message digest to reflect the new value. A MAC solves this problem by incorporating a secret key shared only between Alice and Bob, adding the secret key into the message digest calculation. Thus, if Eve attempts to change either the document or the message digest, without the secret key it will be impossible for her to succeed; Bob will see that the document and/or message digest hash has been tampered with.

Message Authentication Codes are covered in Chapter 4, Section 4.3. The first argument is the algorithm to use, the second argument is a Base64-encoded secret key that is known to both the sender and receiver, and the third argument is the plain text document. The result is a Base64-encoded message digest hash.

The sample code provided with the `PBCrypto` solution demonstrating this method can be found in the `w_generate_mac.cb_generate.clicked()` method. The following example uses this code:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto
String ls_result

IF lnv_crypto.of_getInstance(pbcrypto) AND (Len(sle_secretKey.text) > 0)
THEN
      ls_result =
            pbcrypto.generateMessageAuthenticationCode("HMACMD5",
                  sle_secretKey.text,
                  sle_hashcode.text)
ELSE
      MessageBox("Alert", "You must generate a secret key first!")
END IF
```

For complete details on what algorithms are available for use in the first and second arguments, review Chapter 1, Sections 1.4 through 1.6.

## GenerateMessageDigest

generatemessagedigest ( string algorithm, string document ) returns string

Message digests help in determining whether a document has changed over time. By definition, changing a single bit of a document should result in a radically different message digest value. The two prominent message digest algorithms in use are MD-5 and SHA-1. Message digests are covered in depth in Chapter 4. The first argument is the algorithm to use, and the second argument is the plain text document. The result is a Base64-encoded message digest hash.

The sample code provided with the `PBCrypto` solution demonstrating this method can be found in the `w_generate_message_digest.cb_generate.clicked()` method. The following example uses this code:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto
String ls_result

IF lnv_crypto.of_getInstance(pbcrypto) THEN
      ls_result = pbcrypto.generateMessageDigest("SHA-1",
                        sle_hashcode.text)
END IF
```

For complete details on what algorithms are available for use in the first and second arguments, review Chapter 1, Sections 1.4 through 1.6.

## GenerateSecretKey

generatesecretkey ( string algorithm ) returns string
    throws nosuchalgorithmexception

The safest way to generate a symmetric key is to have a cryptographically secure pseudorandom number generator build a key. This method relies on the JCE `KeyGenerator` engine to create a secure secret key that can be used in a symmetric block cipher. Secret key generation is discussed in Chapter 2, Sections 2.1 through 2.4.

The sample code provided with the `PBCrypto` solution demonstrating this method can be found in the `w_symmetric_encryption.cb_secretkey.clicked()` method. The following example uses this code:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto

TRY
      IF lnv_crypto.of_getInstance(pbcrypto) THEN
            sle_secretkey.text = pbcrypto.generateSecretKey("HMACMD5")
      END IF
CATCH (NoSuchAlgorithmException nsae)
      MessageBox("Alert", nsae.getMessage())
END TRY
```

For complete details on what algorithms are available for use in the first argument, review Chapter 1, Sections 1.4 through 1.6.

## VerifySignature

verifysignature ( string keyAlgorithm, string sigAlgorithm, string pubKey, string document, string signature)  returns Boolean

This method is directly related to the `generateDigitalSignature()` method covered earlier in this paper. For chapter and sample information, please review that method. The first and second arguments contain the key and signature algorithms, respectively. The third argument is the Base64-encoded representation of the RSA public key, the fourth argument is the plain text body of the document that is to be signed, and the fifth argument is the Base64-encoded digital signature to be verified. The result is a Boolean, true if the document has been successfully verified and should be considered valid.

The following example shows how to invoke this method:

```
n_cst_cryptography lnv_crypto
PBCrypto pbcrypto
boolean lb_valid

TRY
      IF lnv_crypto.of_getInstance(pbcrypto) THEN
            lb_valid =
                  pbcrypto.verifySignature("RSA",
                        "MD5withRSA",
                        mle_public.text,
                        mle_message.text,
                        sle_signature.text)
            MessageBox("Signature Verification Results", lb_valid)
      END IF
CATCH (NoSuchAlgorithmException nsae)
      MessageBox("Alert", nsae.getMessage())
END TRY
```

When verifying a digital signature, it is imperative that you use the same key and signature algorithms that generated the digital signature value passed in the fifth argument.

## Conclusion

Cryptography is a topic that cannot be covered adequately in a single white paper. In combination, the demo application, this white paper, the book that is referenced throughout, *Java Cryptography Extensions: Practical Guide for Programmers*, and the Java Cryptography Extensions create a complete package of information that should empower developers to build complex and robust cryptographic solutions into their PowerBuilder applications. Remember: Be safe out there!

**Sybase, Inc.**
**Worldwide Headquarters**
**One Sybase Drive**
**Dublin, CA 94568-7902 USA**
**Tel: +800 8 SYBASE**
**www.sybase.com**