Field Level Encryption

Getting Started with the PKWARE Smart Encryption API

- C++
- .NET
- Java

PKWARE Inc.

Contents

Securing Your Data: An Overview

The PKWARE Smart Encryption Application Programming Interface (API) offers you and your development team several ways to encrypt and secure data. This guide will help you understand and use the tools available to you, You'll also see where to look for more advanced information.

This guide starts with an overview of the principles underlying the encryption tools included with the Smart Encryption API:

- Passphrase-based encryption
- X.509 certificate-based encryption
- Encrypting structured data
- Using digital signatures

Following this overview, find the sections that apply to the programming language(s) you write with. You'll find samples to learn the basic processes to sign and encrypt data.

Much of the reference information here is from the US National Institute of Standards and Technology. The NIST Computer Security Resource Center web site, http://csrc.ncsl.nist.gov/, contains FAQs and documentation relating to computer security. The PKWARE web site, www.pkware.com, also contains information relating to security in SecureZIP.

Basic Passphrase Encryption

Encryption provides confidentiality for data. Unencrypted data is called *plaintext*. Encryption transforms the plaintext data into an unreadable form, called *ciphertext*, using an encryption key. Decryption transforms the ciphertext back into plaintext using a decryption key.

A passphrase uses letters, numbers, spaces and other non-alphanumeric symbols to allow your recipient to open your encrypted file or message. The passphrase used to encrypt a file with the PKWARE Smart Encryption API may be from 1 to 260 characters in length. Files are encrypted using either the 3DES or Advanced Encryption Standard (AES) algorithm. If you use a passphrase to encrypt, anyone who knows the passphrase can decrypt.

Different passphrases may be used for various files, although only one passphrase may be specified per run. To maintain the confidentiality of the data encrypted by a key, the key must be known only by the entities that are authorized to access the data.

The passphrase is not stored, and as a result, care must be taken to keep passphrases secure and accessible by some other source.

Password vs Passphrase?

> **Far too many people choose not-very-secure passwords. These passwords are either too easy for others to guess (repeated public shaming has not kept dictionary words like *password* or standard number sequences like 123456 from being the most popular passwords), too hard for you to remember, or used everywhere (meaning if you know one password, you just might know them all). For these reasons, many security professionals use the term *passphra se* when referring to symmetric keys. This guide follows that convention.**
>
> **What makes a pass*phrase* different from a pass*word*? There is no dictionary definition, but in the English language, phrase suggests "multiple words that go together, but do not form a compound word." An effective passphrase could combine a flavor or seasoning with a farm implement, or an adverb. The longer the passphrase, the more secure it is. Thus you could string all these together: *flavoragriculturalimplementseasoningadverb*. Stronger still with numbers or non-alphanumeric characters, preferably not in spots that separates the "words."**

Encrypting Structured Data

One reason organizations are reluctant to encrypt the contents of individual database fields is that the encrypted version of a field often has many more characters than the original, unencrypted data. This can lead to changing field length limits or data types in ways that would not otherwise make sense.

The PKWARE Smart Encryption API allows applications to encrypt and decrypt structured data in one or more database fields without changing the length of the fields. The class uses strong encryption with the ***Advanced Encryption Standard (AES)*** algo rithm in Cipher Feedback (CFB) mode so that other encryption tools should be able to decrypt the data encrypted with the Smart Encryption API. See "Introduction to Field Level Encryption" for more information on this process.

Confirming Identity through Digital Signatures

A digital signature is an unforgeable mechanism that ensures that the file to which it is attached originates from the owner of the signature and is unchanged since it was signed. The private key from a user's digital certificate is used to attach a digital signature. The signature is *authenticated* by application of the public key from the certificate.

Authentication is a separate operation from data encryption. Whereas encryption is concerned with preventing parties from accessing sensitive data (such as private medical or financial information), authentication confirms that information actually comes unchanged from the purported source.

Authenticating digitally signed data both verifies the signature and validates the signed data.

Validating Digital Signatures

The Smart Encryption API makes use of certificate-based encryption within the public key infrastructure (PKI) to generate and validate digital signatures.

The API provides a means to access the supported (X.509) certificate keys necessary for signing and authentication.

Be aware of these relevant details in Smart Encryption API support for digital signatures.

- The Smart Encryption API supports only RSA keys for X.509 certificate signatures.
- PKI provides an authentication chain for X.509 certificates to guarantee that the signature was created by the purported source.
- Additional facets of validating a certificate's viability for use include a defined range of dates within which a certificate may be used and whether the certificate has been declared to have been revoked.

Introduction to Field Level Encryption

This guide will help you understand and implement field level encryption through the Smart Encryption API.

The API supports field level encryption in these languages:

- Java
- C++
- C#

Syntax will differ among these languages, but this guide will focus on the tasks. See the generated Help system for language-specific information.

What is Field Level Encryption?

Field level encryption (FLE) helps to ensure you can encrypt structured data while preserving its length (the number of characters in the table cell) and data type. If you have a database column filled with nine-digit US Social Security numbers, standard strong encryption adds characters to hide real data. Too often, this forces your organization to choose between security and a bloated database, as the encrypted data violates any character limits you have for database cells. FLE resolves this dilemma, because it will replace those nine digits with nine other characters (including non-printing characters).

Use the Smart Encryption API whenever you need to protect personally identifiable data (including credit card numbers, medical patient numbers, and Social Security numbers) stored in a database.

If you need security with more convenience, this API also allows you to preserve the layout/format of this data as well. See “Int roduction to Format Preserving Encryption” later in this document.

In the next sections, you’ll learn how to use the Smart Encryption API in the supported programming languages.

Field Level Encryption: A Guide for Java and C# Developers

Encrypting an Object

Create an encryption object

1) (Java only) Create a length preserving encryption object:

   public LengthPreservingEncryption()

2) Identify the encryption key you want to use:
   a) Use an existing key
   b) Generate a random encryption key with ***generateRandomKey128***, ***generateRandomKey192***, or ***generateRandomKey256***. The longer the key, the stronger the encryption. Syntax will vary, depending on your language. You will need to store this key somewhere, such as a separate column in your database table. Note that if you store the key in the database, you must encrypt the key.
   c) Derive key from passphrase
3) Generate a random initialization vector (IV) with ***generateRandomIV***. An IV is a random set of characters included in the encrypted file that does not allow an attacker to infer relationships between segments of the encrypted data. You will need to store this IV somewhere, such as a separate column in your database table.
4) To encrypt any byte array, use ***encrypt***:

   Public static byte[] encrypt (byte[] key,
   Byte[] iv,

Byte[] data)

If you don't need to preserve lengths in your encrypted objects, you may also use *encryptString*, *encryptStringBase64*, **or** *encryptStream* **with keys and IVs in place of** *encrypt*. **See the Method Reference for more information.**

Decrypting an Object

Use ***decrypt*** to allow your authenticated users to work with the actual data.

1) Identify the encryption key used to encrypt this data with either of these methods:
   a) Use an existing key
   b) Derive key from passphrase
2) Identify the Initialization Vector (IV). An IV is a random set of characters included in the encrypted file that does not allow an attacker to infer relationships between segments of the encrypted data. The IV is generated when encrypting and stored in an accessible place. You must use the same IV used to encrypt the data you want to decrypt.
3) Use ***decrypt*** to view the encrypted data:

Public static byte[] decrypt (byte[] key,
Byte[] iv,
Byte[] data)

Method Reference

The following table contains the available methods for the array encryption/decryption class for Field Level Encryption. The table shows the Java methods. If you work with C#, you will find that the commands are similar, with different capitalizations. Use the Help system to clarify any issues.

Method	Description	Parameters
generateRandomKey128	Generate random key material for AES128	
generateRandomKey192	Generate random key material for AES192	
generateRandomKey256	Generate random key material for AES256	
generateRandomIV	Returns random initialization vector (IV)	
encrypt	Encrypt byte array using specified key material and IV	***key*** - Key material, array must be either 16 bytes, 24 bytes, or 32 bytes ***iv*** - 16 byte initialization vector ***data*** - Data to encrypt
decrypt	Decrypt byte array using specified key material and IV	***key*** - Key material, array must be either 16 bytes, 24 bytes, or 32 bytes ***iv*** - 16 byte initialization vector ***data*** - Data to decrypt
encryptString	Encrypt a string converted to UTF-8 using the specified key material and IV to a byte array. Depending on the original character set, this method may preserve the original length.	***key*** - Key material, array must be either 16 bytes, 24 bytes, or 32 bytes ***iv*** - 16 byte initialization vector ***data*** - String to encrypt
decryptString	Decrypt a string using the specified key material and IV to a byte array	***key*** - Key material, array must be either 16 bytes, 24 bytes, or 32 bytes ***iv*** - 16 byte initialization vector ***data*** - String to decrypt
encryptStringBase64	Encrypt a string converted to UTF-8 using the specified key material and IV and return the encrypted data as a Base64 string	***key*** - Key material, array must be either 16 bytes, 24 bytes, or 32 bytes

		iv **-** 16 byte initialization vector ***data*** **-** String to encrypt
decryptStringBase64	Decrypt a string using the specified key material and IV	***key*** - Key material, array must be either 16 bytes, 24 bytes, or 32 bytes ***iv*** - 16 byte initialization vector ***data*** - String to decrypt
encryptStream	Creates a stream that will encrypt data as it is written to the stream (Java only)	***key*** - Key material, array must be either 16 bytes, 24 bytes, or 32 bytes ***iv*** - 16 byte initialization vector ***outputStream*** - Stream where the encrypted data will be written
CreateEncryptionStream	(C#) Creates stream that encrypts everything that is read from it or written to it.	***stream*** **-** The stream on which to perform the cryptographic transformation. ***mode*** - Indicates if stream is intended to be read from or written to. ***key*** - Key material ***iv*** - Initialization vector
decryptStream	Creates a stream that will decrypt data as it is read from the stream (Java only)	***key*** - Key material, array must be either 16 bytes, 24 bytes, or 32 bytes ***iv*** - 16 byte initialization vector ***inputStream*** - Stream where the encrypted data will be read
CreateDecryptionStream	(C#) Creates stream that decrypts everything that is read from it or written to it.	***stream*** **-** The stream on which to perform the cryptographic transformation. ***mode*** - Indicates if stream is intended to be read from or written to. ***key*** - Key material ***iv*** - Initialization vector
deriveKey256	Derive key material for AES 256 from the password using PBKDF2	***password*** - Password used to derive the key ***salt*** - Salt used to derive the key. Must be at least 8 bytes. ***iterations*** - Number of iterations for the operation. Must be greater than 0. Default: 50000
deriveKey192	Derive key material for AES 192 from the password using PBKDF2	***password*** - Password used to derive the key ***salt*** - Salt used to derive the key. Must be at least 8 bytes. ***iterations*** - Number of iterations for the operation. Must be greater than 0. Default: 50000
deriveKey128	Derive key material for AES 128 from the password using PBKDF2	***password*** - Password used to derive the key ***salt*** - Salt used to derive the key. Must be at least 8 bytes. ***iterations*** - Number of iterations for the operation. Must be greater than 0. Default: 50000

Field Level Encryption: A Guide for C++ Developers

Encrypting an Object

Create an encryption object

Use this command to create a length preserving encryption object:

> **extern PKSymmetricEncryptionPtr pkSymmetricEncryption(PKSession* pSession, PKKeyMaterial* pKeyMaterial, LPCBYTE pbIV = NULL, PKUINT32 cbIV = 0, PKUINT32 alg = CALG_AES_256, PKUINT32 mode = PK_ENCRYPT_MODE_CFB);**

The ***pSession*** parameter creates the object, and ***pKeyMaterial*** identifies the encryption key. Choose the strength of your encryption algorithm (***CALG_AES_128***, ***CALG_AES_192***, or ***CALG_AES_256***) with the ***alg*** parameter.

Select an encryption mode from this list:

- Cipher Block Chaining (***PK_ENCRYPT_MODE_CBC***)
- Cipher Feedback (***PK_ENCRYPT_MODE_CFB***)

If you choose to use an initialization vector (***IV***), you need to identify this vector (***pbIV***) and its size (***cbIV***). An IV is a random set of characters included in the encrypted file that does not allow an attacker to infer relationships between segments of the encrypted data.

Encrypt the Byte Array

There are several ways to encrypt a byte array, described in the help. To use ***encrypt***:

virtual PKUINT32 encrypt(LPCBYTE pbInput, PKUINT32 cbInput, PKBuffer& output) = 0;

Decrypting an Object

Use ***decrypt*** in the same manner as above to allow your authenticated users to work with the actual data. When decrypting, one must use the same key and initialization vector as was used for encryption.

Method Reference: PKSymmetricEncryption

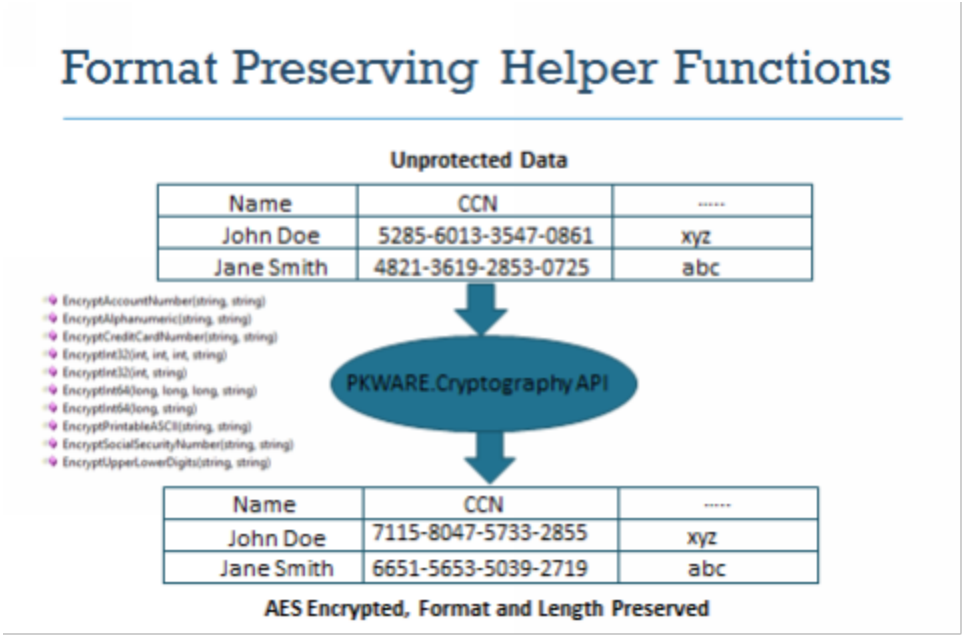
Method	Description	Parameters
encrypt	Encrypt byte array	***pbInput*** **-** Data to encrypt ***cbInput*** **-** Size of data to encrypt ***output*** **-** Output buffer to receive encrypted data
decrypt	Decrypt byte array	***pbInput*** **-** Data to decrypt ***cbInput*** **-** Size of data to decrypt ***output*** **-** Output buffer to receive decrypted data
encryptBase64	Encrypt byte array and encode as Base64	***pbInput*** **-** Data to encrypt ***cbInput*** **-** Size of data to encrypt ***output*** **-** Output buffer to receive encrypted and encoded data
decryptBase64	Decrypt a string using the specified key material and IV	***pbInput*** **-** Data to decode and decrypt ***cbInput*** **-** Size of data to decode and decrypt ***output*** **-** Output buffer to receive decoded/decrypted data
PKSymmetricEncryptionPtr	Smart pointer to the symmetric encryption/decryption object	***pSession*** **-** Session object ***pKeyMaterial*** **-** Key material ***pbIV*** **-** Initialization Vector (optional) ***cbIV*** **-** Initialization vector size (optional) ***alg*** **-** Algorithm ***mode*** **-** Encryption mode

Introduction to Format Preserving Encryption

Every day, thousands of people visit your company's website to do business and access information. Financial transactions and other interactions with personally identifiable private information pass through constantly. Your online store allows customers to register their personal information, including credit card numbers, addresses, and birthdates. They access their account information with a username and passphrase. All of this information is quite valuable for identity thieves and ordinary thieves.

Encrypting your database files certainly raises the security level, but reduces the performance of your site, when milliseconds count for online customer satisfaction. With the Smart Encryption API, you can focus encryption on specific fields while preserving various aspects of the original data.

Using ***Format Preserving Encryption (FPE)*** foils attackers by displaying the encrypted text (called *ciphertext* by cryptographers) using the same format as the original data being encrypted. That is, a legitimate 16-digit credit card number is replaced by 16 random digits in the database. A separate key is needed to decrypt and use the actual credit card account. See the figure below.



Basics

To add format preserving encryption to your application, call the public class ***FormatPreservingEncryption***. Syntax may vary among the supported languages. Identify the field you wish to encrypt as ***byte []***, and the secret key to encrypt with.

You can use the Smart Encryption API to encrypt and decrypt fields in your application. Your application can run as part of an account management form, and interact with your database.

Better Security with Tweaks

The data will be encrypted or decrypted using the key supplied by the caller when this object is created. The caller should also provide a *tweak value* for each encrypt or decrypt operation. The tweak value should be unique to a field or row to strengthen the encryption.

A tweak is a number or a text string that is used on every encryption or decryption to choose a different permutation of values on each instance. Essentially it converts the common AES key used by all the data to a customized key; a different key for each instance of data. The tweak changes the permutation of values – instead of 10000 becoming 511 for all employees, it may become 6789 for one employee, 12 for another, and 9847 for yet another. By making every employee store its salary with a different tweak value, it means that each employee's data is effectively encrypted with a different key, producing a different permutation. Knowing anything about one permutation does not help an attacker figure out other permutations.

Unlike the AES key, the tweak does not need to be a secret. You can make the tweak completely public and it won't help an attacker. This is because the relationship between a tweak and the permutation that results is wildly unpredictable. Changing even one bit in a tweak produces an altogether new permutation.

The tweak is a way of choosing a random permutation. The more tweaks we use, the more unlikely it is that two permutations are the same. This makes it unlikely that an attacker who gains some knowledge about some of the data can make any good use of that to decipher the rest of the data.

Traditional block ciphers and hashes use long, random strings of data (initialization vector [IV] or salt) in a similar way as we're using a tweak here. The difference between a tweak and a salt is that the salt is chosen randomly on each encryption and stored along with the encrypted value – it is still public knowledge, but not predictable or similar to any previously used salt (we hope). A tweak is also unique (hopefully) but not random – it is reproducibly derived from some intrinsic information about the data being stored and then (internally) generates the same effective random salt value to use as part of the encryption. You supply the tweak – and the library regenerates the salt each time. Since the goal of format preserving encryption is to not make you need more space to store ciphers, a tweak is a better solution than a long random string (IV or salt) which must also be stored with the cipher.

In fact, the best set of tweaks are those where every item you store uses a different tweak, and thus has the strong chance to pick

permutations at random. Further, it is ideal if each time you update a value, you choose a new tweak, so that a given tweak is never used twice, ever. In practice, this may be difficult. If you have nowhere to store a variable tweak in your data model, then you're pretty much only able to construct a tweak from the existing data. However, this is still a great improvement in security.

Choosing a Tweak Value

So how do you go about choosing a tweak? Well, if every row in a database has a different row id (id) and every column has its own table name and column name, then a concatenation of those three things is an excellent tweak! Example: For a table called Customers, column called "CreditCard", row id 33, a perfectly great tweak is "Customers.CreditCard.33". In most real-world scenarios, this is easily computed for any given use case.

Example: Securing a Customer Database

Your online store allows customers to register their personal information, including credit card numbers, addresses, and birthdates. They access their account information with a username and passphrase.

Create an encryption object

1. Create a format preserving encryption object:

   Public FormatPreservingEncryption (byte[] key)

   You can define the ***key*** value, or use the SHA1 algorithm by declaring the ***key*** value null.

1. Call ***encryptCreditCardNumber*** with the two required parameters:
   - ***value*** is the unencrypted credit card number
   - Add the ***tweak*** value. This should differ with each record or field (see "Choosing a Tweak Value"), but once defined should never change.

1. The function will verify the *Luhn checksum* to ensure that the number entered is a valid credit card number, then encrypt the data. It will return an encrypted value in the same format, with a valid Luhn checksum.

Decrypting an object

Use ***decryptCreditCardNumber*** in the same manner as above to allow your authenticated users to work with the actual credit card number. The tweak value must be identical to the value used by ***encryptCreditCardNumber*** to successfully decrypt.

This method will also confirm the accuracy of the credit card number with the Luhn checksum. An exception is thrown if the entered number does not match.

Method Reference

FormatPreservingEncryption Methods

The following table contains the available methods for the FormatPreservingEncryption class:

Method	Description	Parameters
decryptAccountNumber	Decrypts ASCII digits within the string, replacing each digit with another digit. Characters outside this range will not be decrypted.	***value*** - String containing data to decrypt ***tweak*** - The encryption tweak, this value should be changed for each record or field
decryptAlphanumeric	Decrypts the ASCII uppercase, lowercase and digits within the string replacing each character with another from one of these ranges. For example an uppercase letter could be replaced with an uppercase character, lowercase character or a digit. Characters outside this range will not be decrypted.	***value*** - String containing data to decrypt ***tweak*** - The encryption tweak, this value should be changed for each record or field
decryptCreditCardNumber	Decrypts the ASCII digits within the string that represents	***value*** - String containing data to

	a credit card number. This function will verify the Luhn checksum prior to decrypting the credit card number. Characters outside this range will not be decrypted.	decrypt ***tweak*** **-** The encryption tweak, this value should be changed for each record or field
decryptInt32	Decrypts the specified 32-bit integer within a specified range OR Decrypts the specified 32-bit integer within the range of 0 and Integer.MAX_VALUE. The range can be between 1 and 2 ^ 31.	***value*** - String containing data to decrypt ***min*** - Minimum value ***max*** - Maximum value ***tweak*** **-** The encryption tweak, this value should be changed for each record or field
decryptInt64	Decrypts the specified 64-bit integer within a specified range OR Decrypts the specified 64-bit integer within the range of 0 and Integer.MAX_VALUE. The range can be between 1 and 2 ^ 63.	***value*** - String containing data to decrypt ***min*** - Minimum value ***max*** - Maximum value ***tweak*** **-** The encryption tweak, this value should be changed for each record or field
decryptPrintableASCII	Decrypts the printable ASCII characters (values 9, 32 - 126) within the string. Each character within this range will be decrypted and replaced by another character in that same range.	***value*** - String containing data to decrypt ***tweak*** **-** The encryption tweak, this value should be changed for each record or field
decryptSocialSecurityNumber	Decrypts the ASCII digits of the Social Security Number (SSN) within the string, replacing each digit with another digit. Characters outside this range will not be decrypted.	***value*** - String containing data to decrypt ***tweak*** **-** The encryption tweak, this value should be changed for each record or field
decryptString	Decrypt the specified value using an array of classifiers to determine which characters in the string should be decrypted.	***value*** - The string containing the data to decrypt. ***tweak*** - The encryption tweak, this value should be changed for each record or field. ***classifierList*** - Array of classifiers
decryptUpperLowerDigits	Decrypts the ASCII uppercase, lowercase and digits within the string replacing each character with another of the same type. For example uppercase letters will be replaced with uppercase letters and digits will be replaced with digits. Characters outside this range will not be decrypted.	***value*** - String containing data to decrypt ***tweak*** **-** The encryption tweak, this value should be changed for each record or field
encryptAccountNumber	Encrypts the ASCII digits within the string, replacing each digit with another digit. Characters outside this range will not be encrypted.	***value*** - String containing data to encrypt ***tweak*** **-** The encryption tweak, this value should be changed for each record or field
encryptAlphanumeric	Encrypts the ASCII uppercase, lowercase and digits within the string replacing each character with another from one of those ranges. For example an uppercase letter could be replaced with an uppercase character, lowercase character or a digit. Characters outside this range will not be encrypted.	***value*** - String containing data to encrypt ***tweak*** **-** The encryption tweak, this value should be changed for each record or field
encryptCreditCardNumber	Encrypts the ASCII digits within the string that represents a credit card number. This function will verify the Luhn checksum prior to encrypting the credit card number and	***value*** - String containing data to encrypt

	will produce an encrypted value with a valid Luhn checksum. Characters outside this range will not be encrypted.	***tweak*** **-** The encryption tweak, this value should be changed for each record or field
encryptInt32	Encrypts the specified 32-bit integer within a specified range OR Encrypts the specified 32-bit integer within the range of 0 and Integer.MAX_VALUE. The range can be between 1 and 2 ^ 31.	***value*** - String containing data to encrypt ***min*** - Minimum value ***max*** - Maximum value ***tweak*** - The encryption tweak, this value should be changed for each record or field
encryptInt64	Encrypts the specified 64-bit integer within a specified range OR Encrypts the specified 64-bit integer within the range of 0 and Integer.MAX_VALUE. The range can be between 1 and 2 ^ 63.	***value*** - String containing data to encrypt ***min*** - Minimum value ***max*** - Maximum value ***tweak*** - The encryption tweak, this value should be changed for each record or field
encryptPrintableASCII	Encrypts the printable ASCII characters (values 9, 32 - 126) within the string. Each character within this range will be encrypted and replaced by another character in that same range.	***value*** - String containing data to encrypt ***tweak*** - The encryption tweak, this value should be changed for each record or field
encryptSocialSecurityNumber	Encrypts the ASCII digits of the Social Security Number (SSN) within the string, replacing each digit with another digit. Characters outside this range will not be encrypted.	***value*** - String containing data to encrypt ***tweak*** - The encryption tweak, this value should be changed for each record or field
encryptString	Encrypt the specified value using an array of classifiers to determine which characters in the string should be encrypted.	***value*** - The string containing the data to encrypt. ***tweak*** - The encryption tweak, this value should be changed for each record or field. ***classifierList*** - Array of classifiers
encryptUpperLowerDigits	Encrypts the ASCII uppercase, lowercase and digits within the string replacing each character with another of the same type. For example uppercase letters will be replaced with uppercase letters and digits will be replaced with digits. Characters outside this range will not be encrypted.	***value*** - String containing data to encrypt ***tweak*** - The encryption tweak, this value should be changed for each record or field

Classifier Interface Methods

In FormatPreservingEncryption, each character of the string is examined to see if it should be changed, and what family of characters it belongs to. This phase is performed by character classifier objects which implement a specific interface called the Classifier interface. In C# it is called IClassifier, following the naming conventions of .NET. In Java, the interface is called Classifier. In C++, there are no interfaces, but the abstract base class is called Classifier.

The following table contains the available methods for the Classifier Interface:

Method	Description	Parameters
getModulus	Returns the total number of possible elements in the set of characters to be encrypted.	
Index	Return the position of the character within the set of possible elements or -1 if the character is not in the set.	***ch*** - Character to be classified

Restore	Returns the character for the specified position within the set of possible characters	***pos*** **-** Character position within the set

Creating and Verifying Digital Signatures: A Guide for .NET and C++ Developers

Use the Smart Encryption API to create and verify X.509 digital signatures using the Cryptographic Message Syntax (CMS, also known as the PKCS#7 standard) or XML files (detached or enveloped).

.NET/C# Method Reference: PKArchive.Net

Method	Description	Parameters
virtual void addSigner **(PKWARE::ArchiveAPI::IPKCertificate^ cert)**	Add signer to the list. SHA-1 hash algorithm used. For XML enveloped signatures, only one signer.	***cert*** - Signer certificate with private key
virtual void addSigner **(PKWARE::ArchiveAPI::IPKCertificate^ cert** **PKWARE::ArchiveAPI::PK_HASH_ALG alg)**	Add signer to the list with a specified hash algorithm. For XML enveloped signatures, only one signer.	***cert*** - Signer certificate with private key ***alg*** - Hash algorithm to use
PKWARE::ArchiveAPI::IPKCertificateCollection^ Store	Certificates associated with signature	
virtual Signer ^ GetSigner (UInt32 idx)	Returns specified signer	***idx*** - Signer index
UInt32 SignerCount	Returns number of signers	
virtual array<Byte> ^ Sign	Creates CMS/PKCS#7 detached, XML enveloped or XML detached signature	***data*** - Data to sign. For XML enveloped signatures this must be UTF-8 string (such as the output of Encoding.UTF8.GetBytes()). This implementation does not resolve external URI, so the data to sign must be provided even for detached XML signatures. ***uri*** - Universal Resource Identifier used only for XML detached signatures. ***flags*** - Flag that indicates type of signature to create ***pinStream*** - Stream to sign. Must contain UTF-8 XML for XML enveloped signatures. In XML signatures, this stream will rewind to its initial position. This implementation does not resolve external URIs, so data to sign must be provided (even for detached XML signatures) ***pOutStream*** - Stream where signature will be written. For XML signatures, this is UTF-8 XML.
virtual Boolean Verify	Verifies CMS/PKCS#7 detached or XML enveloped or detached signature. Signer information loaded. Signer certificate is not validated.	***data*** - Pointer to data to verify. Not used for XML enveloped signatures. This implementation does not resolve external URIs, so data to verify must be provided for detached XML signatures ***signature*** - signature. For XML enveloped signatures, this must be UTF-8 XML string (such as the output of Encoding.UTF8.GetBytes()). ***flags*** - Indicates type of signature (sign, encrypt, both) to verify. ***pinStream*** - Stream to verify. Not used for XML enveloped signatures. In detached XML signatures, this stream will rewind to its initial position. This

		implementation does not resolve external URIs, so data to sign must be provided for detached XML signatures. ***pSignature*** **-** Stream that contains signature data. For XML enveloped or detached signatures, this must be UTF-8 XML.

C++ Method Reference: PKSignVerify

Method	Description	Parameters
Virtual PKBOOL PKSignVerify::addSigner	Add signer to the list. For XML enveloped signatures, only one signer.	***pCert*** - Signer certificate (w/private key) ***hashAlgID*** - Hash algorithm to use
Virtual PKCertificateStorePtr PKSignVerify::getCertificateStore	Returns certificate store associated with signature.	
virtual PKCertificatePtr PKSignVerify::getSigner	Returns signer information	***idx*** - Signer index ***pHashAlgID*** - Returns the signer's associated hash algorithm if not NULL ***pStatus*** - Returns the signer's status if not NULL
virtual PKUINT32 PKSignVerify::getSignerCount	Returns number of signers	
virtual void PKSignVerify::sign	Creates CMS/PKCS#7 detached, XML enveloped or XML detached signature	***pbin*** - Pointer to data to sign, cannot be NULL. For XML enveloped signatures this must be UTF-8 XML string. This implementation does not resolve external URI, so the data to sign must be provided even for detached XML signatures. ***cbin*** - Size of the data to sign. For XML enveloped signature this must be the length of the XML string not counting terminating '\0' ***signature*** - Returned signature. For XML signatures this is UTF-8 XML string without terminating '\0' ***szURI*** - Universal Resource Identifier used only for XML detached signatures. ***flags*** - Flag that indicates type of signature to create ***pinStream*** - Stream to sign (cannot be NULL). Must contain UTF-8 XML for XML enveloped signatures. In XML signatures, this stream will rewind to its initial position. This implementation does not resolve external URIs, so data to sign must be provided (even for detached XML signatures) ***pOutStream*** - Stream where signature will be written, cannot be NULL. For XML signatures, this is UTF-8 XML.
Virtual PKBOOL PKSignVerify::verify	Verifies CMS/PKCS#7 detached or XML enveloped or detached signature. Signer information loaded. Signer certificate is not validated.	***pbin*** - Pointer to data to verify. Not used for XML enveloped signatures. This implementation does not resolve external URIs, so data to verify must be provided for detached XML signatures ***cbin*** - Size of the data to verify. Not used for XML enveloped signatures. ***pbSignature*** - Pointer to signature, cannot be NULL. For XML enveloped signatures, this must be UTF-8 XML string. ***cbSignature*** - Size of the signature. For XML signature, this must be the length of the XML string, not counting the

		terminating '\0'. ***Flags*** **-** Indicates type of signature (sign, encrypt, both) to verify. ***pinStream*** **-** Stream to verify. Not used for XML enveloped signatures. In XML signatures, this stream will rewind to its initial position. This implementation does not resolve external URIs, so data to sign must be provided (even for detached XML signatures). ***pSignature*** **-** Stream that contains signature data, cannot be NULL. For XML enveloped signatures, this must be UTF-8 XML.

Creating and Verifying Digital Signatures: A Guide for Java Developers

Use the Smart Encryption API to create and verify X.509 digital signatures using the Cryptographic Message Syntax (CMS, also known as the PKCS#7 standard) or XML files (detached or enveloped).

Method Reference: Java

Method	Description	Parameters
addSigner (CMS and XML Detached)	Add signer and associated private key that will be used to sign the data. SHA-1 hash algorithm used. For XML enveloped signatures, only one signer.	***certificate*** - X.509 certificate associated with the signature ***privateKey*** - Private key that corresponds to the X.509 certificate ***hashID*** - Hash algorithm to use ***certificateChain*** - Optional certificate chain to verify the X.509 certificate
setSigner (XML Enveloped)	Add signer and associated private key that will be used to sign the data. For XML enveloped signatures, only one signer.	***certificate*** - X.509 certificate associated with the signature ***privateKey*** - Private key that corresponds to the X.509 certificate ***hashID*** - Hash algorithm to use, possible values are: ***ArchiveEntry.HASH_SHA1, ArchiveEntry.HASH_SHA_256, ArchiveEntry.HASH_SHA_384*** and ***ArchiveEntry.HASH_SHA_512*** ***certificateChain*** - Optional certificate chain to verify the X.509 certificate
getSignerCount (CMS and XML Detached)	Returns number of signers	
getSigner (CMS and XML Detached)	Returns signer at the specified position	***idx*** - index position within the signer array
getCertificates	Additional X.5090 certificates associated with the CMS signature	
sign (XML Detached)	Create a detached XML signature for the item identified by the URI	***baseUri*** - Base URI for verifying the URI within the detached XML signature. For a file-based URI, this will be a folder where the source file is located; in the form "file:// folder/". ***uri*** - zthe item to sign, typically a file or HTTP(S) address.
sign (XML Enveloped)	***public byte[] sign(byte[] data)*** Create an enveloped XML signature for the supplied XML data byte array. ***public byte[] sign(java.io.InputStream inputStream)*** Create an enveloped XML signature for the supplied XML data stream	***data*** - Input byte array with the XML to sign ***inputStream*** - Input data stream with the XML to sign
signData	Create CMS detached signature for the byte array	***data*** - Byte array to sign
signFile	Create CMS detached signature for the specified file	***file*** - The file to sign
signStream	Create CMS signed output stream that will create CMS signature for data as it is written to the returned stream	***signatureStream*** - Output stream that will hold the detached signature
verify	***public boolean verify(java.lang.String***	***baseUri*** - Base URI for verifying the URI

(XMLVerifyDetached)	***baseUri,*** ***byte[] data)*** Verify the detached XML signature within the byte array ***public boolean verify(java.lang.String baseUri,*** ***java.io.InputStream inputStream)*** Verify the detached XML signature within the input stream	within the detached XML signature. For a file-based URI, this will be a folder where the source file is located; in the form "file:// folder/". ***data*** - Input byte array with the XML to verify ***inputStream*** - Input data stream with the XML to verify
verify (XMLVerifyEnveloped)	***public boolean verify(byte[] data)*** Verify the detached XML signature within the byte array ***public boolean verify(java.io.InputStream inputStream)*** Verify the detached XML signature within the input stream	***baseUri*** - Base URI for verifying the URI within the detached XML signature. For a file-based URI, this will be a folder where the source file is located; in the form "file:// folder/". ***data*** - Input byte array with the XML to verify ***inputStream*** - Input data stream with the XML to verify
verifyData (CMS verify)	Verify the CMS signature on a byte array	***data*** - The data that was originally signed ***signature*** - The CMS detached signature
verifyFIle (CMS verify)	Verify the CMS signature on a file	***dataFile*** - The data file that was originally signed ***signature*** - The CMS detached signature

Glossary of Cryptographic Terms

Advanced Encryption Standard (AES): The official US Government encryption standard for customer data. This algorithm requires one of three key strengths: 128-bit, 192-bit, or 256-bit.

Format Preserving Encryption (FPE): Encrypting a set of data so that the encrypted output appears in the same format as the original, plaintext data.

Luhn checksum: A formula commonly used to confirm that a user typed a credit card, identification, or other account number correctly into a form. The formula verifies a number against its included check digit, which is usually appended to a partial account number to generate the full account number.

User Help and Contact Information

For licensing, please contact Sales at 937-847-2374 (888-4PKWARE / 888-475-9273) or email pksales@pkware.com.

For technical assistance, contact Technical Support at 937-847-2687 or visit the support web site: https://www.pkware.com/support.