

[print](#) | [close](#)

APIs by Example: AES Encryption to Actual Field Length

[System iNetwork Programming Tips Newsletter](#)

[Carsten Flensburg](#)

Carsten Flensburg

Thu, 08/28/2008 (All day)

I've discussed and demonstrated the Cryptographic Services APIs in past issues of this newsletter. The encryption examples that I presented have all used the Advanced Encryption Standard (AES) algorithm, which offers a current, strong, and well-performing encryption method that has been [adopted](#) as an encryption standard by the U.S. government. If you have followed those examples, you know that AES encryption is a [block cipher](#) and that the ciphertext produced by AES therefore always is an exact multiple of the block size applied to the cryptographic process. The block size specified for the AES algorithm is 16 bytes, although the Cryptographic Services APIs' implementation of AES offers 24 and 32 bytes block sizes as well.

For some purposes and situations, the block-size determined length of the ciphertext could be regarded as an inconvenience. In today's APIs by Example, I explain a method for handling data not a multiple of the block length.

If you for example have an existing database and want to encrypt one or more fields in this database, you could be faced with the requirement to increase the size of these fields or find alternate solutions to store the ciphertext. This would be the case if the fields' current size was not an exact multiple of the encryption block size, or if the field's data type was incapable of storing a cipher string, as for example a packed numeric field. For character fields there's however another alternative, as pointed out in a recently released [Redbook](#) that introduces the concept of the CUSP (Cryptographic Unit Support Program) operation mode in a System i context.

The Redbook *IBM System i Security: Protecting i5/OS Data with Encryption* was finalized and published on July 24, 2008. I've provided links to the Redbook at the end of this article. The Redbook is written by IBM and industry cryptography specialists and offers a multitude of information and discussions relating to introduction to as well as planning and implementation of data encryption. For anyone facing an encryption project, this Redbook is definitely mandatory reading. The Redbook covers fundamental encryption knowledge and principles, step-by-step implementation guidelines, and lots of sample code and programming examples. And, as I mentioned, there's also a discussion of a method that can be applied to common block cipher encryption algorithms, in order to preserve the length of the cleartext data in its encrypted state.

This method is referred to as CUSP operation mode. I first noticed the CUSP mode being mentioned when I browsed the Cryptographic Services APIs release 6.1 documentation of the enhancements provided with this release to the encryption and decryption APIs, CUSP mode being one of them. The manual describes CUSP mode in the following wording:

- CUSP is a special type of CBC mode documented in the z/OS ICSF Application Programmer's Guide (SA22-7522). It is used for handling data not a multiple of the block length. The length of encrypted data in CUSP mode will always equal the length of plaintext data.

At that point, I didn't fully comprehend the implications of the above statement and since I do not yet have access to a 6.1 system, I didn't find any cause to pursue the topic any further. When perusing the aforementioned encryption Redbook, however, I came across the CUSP topic once again. Here's the part from a section discussing the AES algorithm that caught my interest:

Use CBC mode to hide patterns in the data. If the size of your data is not on a block size boundary, you have the following options.

- Select the pad option. This increases the size of your ciphertext to the next block-size multiple.
- To keep your ciphertext the same size as your plaintext, implement CUSP mode of operation. CUSP mode can be implemented as follows:

```
a) Use CBC mode to encrypt up to the last partial block of
plaintext.
b) Encrypt the last block of ciphertext again.
c) XOR the last partial block of plaintext data with the same
number of bytes from step b.
```

If the data length is less than the block size, the initialization vector is used in place of the last block of ciphertext in step a. This process is identical for encrypting or decrypting.

While it undoubtedly is pretty straightforward to take advantage of the built-in CUSP mode on 6.1 systems, following the above explanation and instructions it is also possible to implement the length preserving CUSP mode as a do-it-yourself challenge on earlier releases. To go down that route and to get even more details about the CUSP mode of operation the *z/OS V1R9.0 Cryptographic Services ICSF Application Programmer's Guide* mentioned in the above API manual is certainly worth consulting. I've included a link to this information at the end of the article for your convenience in case you want to have a closer look. Or, if you simply want to get to the conclusion, here's my interpretation of the programming steps involved in implementing the CUSP mode of operation – first the encryption process:

1. Determine the full length of the data to encrypt. Since you are going to end up with a ciphertext of equal size to the input data, it is important that you use the field length and not the actual size of the field value. Doing the latter would expose the length of the field value, and revealing such information would be considered a leak in terms of confidentiality as well as implementation.
2. Determine the encryption block size and divide the input data string into two parts. The first part is made up by the maximum number of full data blocks available. The second part is made up of the partial block size remaining. If the input data string length is an exact multiple of the block size, the second part will be null and steps 4 to 6 can be skipped and the result of step 3 stored as the final result of the encryption process.
3. Encrypt the first part of the divided input data string using the AES algorithm, the encryption key, no padding and CBC mode (initialization vector (IV) used). Store the IV so that it can be safely retrieved when needed for the decryption process.
4. Extract the last full block of cipher text and encrypt this part again using the AES algorithm, the same encryption key, no padding and ECB mode (no initialization vector used). If the full length of the input data string is less than the encryption block size, the IV from step 3 is used as input to this step.

5. XOR (eXclusive Or) the second part of the input data string found in step 2 with a equal sized substring of the ciphertext produced in step 4.
6. Append the result of the XOR in step 5 to the ciphertext produced in step 3 and store the full ciphertext string as the final result of the CUSP mode encryption process.

To restore the plaintext value from the ciphertext produced in the encryption process, the following steps, which basically are a reversal of the steps above, are involved:

1. Determine the full length of the data to decrypt. For ciphertext stored in a file, this would normally be the field size.
2. Determine the encryption block size and divide the input ciphertext into two parts. The first part is made up by the maximum number of full data blocks available. The second part is made up of the partial block size remaining. If the input ciphertext length is an exact multiple of the block size, the second part will be null and steps 4 to 6 can be skipped and the result of step 3 stored as the final result of the decryption process.
3. Decrypt the first part of the divided input ciphertext using the AES algorithm, the encryption key, no padding and CBC mode (initialization vector (IV) used). The IV must be the exact same as used in the encryption process.
4. Extract the last full block of ciphertext and encrypt this part again using the AES algorithm, the same encryption key, no padding and ECB mode (no initialization vector used). If the full length of the input data string is less than the encryption block size, the IV from step 3 is used as input to this step.
5. XOR (eXclusive Or) the second part of the ciphertext found in step 2 with a equal sized substring of the ciphertext produced in step 4.
6. Append the result of the XOR in step 5 to the plaintext produced in step 3 and return the full plaintext string as the final result of the CUSP mode decryption process.

To demonstrate the process I've written a short test program implementing the CUSP mode of operation in accordance with the directions stated above. When you run the test program it will prompt you to specify a text string of arbitrary length. The test program then takes your input data and then performs all the required encryption processing to first produce an encrypted string of the same length as the input data, and then subsequently reverse the process and decrypt the ciphertext and return the plaintext value. All steps are displayed on the screen.

If you browse the program source code you'll notice that I've encapsulated all the code that relates to the CUSP mode of operation in the *Apply CUSP mode - ApyCuspMode()* subprocedure, which in turn relies on the *Block input data - BlkInpDta()* and *Get final data block - GetFinBlk()* subprocedures. The former divides the input data string into the two parts described in step 2 above, the latter extracts the last full data block as described in step 4. Once this part of the recipe is taken care of, the rest of the work is pretty straight forward. If you run the test program in the source debugger, you can follow the individual steps as the code is executed. Use function key F10 to step through the program one single statement at the time, and use F22 to step into the subprocedures.

A couple of practical notes: Since I have no way of knowing exactly how the built-in release 6.1 CUSP mode is implemented, I would advise you to assume that the ciphertext produced by the method presented here not necessarily will be correctly decrypted by the built-in CUSP mode and vice versa. So to change from one implementation to another a conversion step might be required.

Storing ciphertext in a normal file character field could lead to potential problems if the file CCSID attribute is changed or data is copied to other files. You should therefore add the field keyword CCSID(66535) to the field storing the ciphertext to ensure that no unwanted data conversion is performed. Using the Change Physical File (CHGPF) command specifying the source file containing the altered source member will let you perform this change quite easily. Adding the CCSID keyword will not change the file's record format identifier and therefore not cause any program file level checks.

This APIs by Example includes the following sources:

```
CBX195    -- RPGLE    -- Encrypt and decrypt data - services
CBX195B   -- SRVSRCL -- Encrypt and decrypt data - binder source
CBX195T   -- RPGLE    -- Encrypt and decrypt data - test
```

To create all above objects, follow the instructions in the respective source headers. To subsequently run the test program type the following command on a command line and press Enter:

```
CALL    CBX195T
```

Then follow the instructions and watch the CUSP mode encryption and decryption process as it unfolds.

Previously published related articles:

APIs by Example, November 8, 2007:

Cryptographic Key Management - Loading and Setting Master Keys:

<http://systeminetwork.com/article/apis-example-cryptographic-key-management-loading-and-setting-master-keys>

APIs by Example, December 13, 2007:

Cryptographic Key Management - Testing and Clearing Master Keys:

<http://systeminetwork.com/article/apis-example-cryptographic-key-management-testing-and-clearing-master-keys>

APIs by Example, January 24, 2008:

Cryptographic Key Management – Creating and Translating Key Stores:

<http://systeminetwork.com/article/apis-example-cryptographic-key-management-creating-and-translating-key-stores>

APIs by Example, February 28, 2008:

Cryptographic Key Management – Creating, Displaying, and Deleting Key Records:

<http://systeminetwork.com/article/apis-example-cryptographic-key-management-%E2%80%93-creating-displaying-and-deleting-key-records>

APIs by Example, March 27, 2008:

Cryptographic Key Management - Creating Data Key Stores and More

<http://systeminetwork.com/article/apis-example-crypto-key-management-creating-data-key-stores-and-more>

APIs by Example, April 24, 2008:

Cryptographic Key Management - Encrypt/Decrypt with Key Hierarchy

<http://systeminetwork.com/article/apis-example-crypto-key-mgmt-encryptdecrypt-key-hierarchy>

IBM Documentation:

z/OS V1R9.0 Cryptographic Services ICSF Application Programmer's Guide (SA22-7522-09): CUSP
<http://publib.boulder.ibm.com/infocenter/zos/v1r9/index.jsp?topic=/com.ibm.zos.r9.e0za100/e0z2a180147.htm>

IBM System i Security: Protecting i5/OS Data with Encryption
<http://www.Redbooks.ibm.com/redbooks/pdfs/sg247399.pdf>

IBM System i Security: Protecting i5/OS Data with Encryption: CUSP mode
<http://www.Redbooks.ibm.com/redbooks/pdfs/sg247399.pdf#page=88>

Encrypt Data (OPM, QC3ENCDT; ILE, Qc3EncryptData) API - release 6.1:
<http://publib.boulder.ibm.com/infocenter/systems/scope/i5os/topic/apis/qc3encdt.htm>

Decrypt Data (OPM, QC3DECDT; ILE, Qc3DecryptData) API – release 6.1
<http://publib.boulder.ibm.com/infocenter/systems/scope/i5os/topic/apis/qc3decdt.htm>

This article demonstrates the following Cryptographic Services APIs:

Encrypt Data (Qc3EncryptData) API:
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qc3encdt.htm>

Decrypt Data (Qc3DecryptData) API:
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qc3decdt.htm>

Generate Symmetric Key (Qc3GenSymmetricKey) API:
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qc3gensk.htm>

Generate Pseudorandom Numbers (Qc3GenPRNs) API:
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qc3genprns.htm>

Create Algorithm Context (Qc3CreateAlgorithmContext) API:
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qc3crtax.htm>

Create Key Context (Qc3CreateKeyContext) API:
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qc3crtkx.htm>

Destroy Algorithm Context (Qc3DestroyAlgorithmContext) API:
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qc3desax.htm>

Destroy Key Context (Qc3DestroyKeyContext) API:
<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qc3deskx.htm>

You can retrieve the source code for this API example from:
http://www.pentontech.com/IBMContent/Documents/article/57114_639_AesCusp.zip

Source URL: <http://iprodeveloper.com/rpg-programming/apis-example-aes-encryption-actual-field-length>