Encryption enhancements: Field procedure support in DB2 for i 7.1

Secure your IBM i applications

Xing Xing Shen  
Jia Tian Zhong  
Shu Pang  
Carol L Ramler  

January 06, 2011

Column-level encryption support is one of the major headline DB2® features in the IBM® i 7.1 release. The DB2 for i 7.1 support enables transparent column-level encryption implementations with a field-level exit routine known as a field procedure. Discover how this exciting support allows developers to more easily and flexibly build a secure application.

Introduction

Security issues have always been in the headlines, but have grown much more important in recent years. Enterprises must comply with industry and government regulations to ensure that their sensitive information such as credit card numbers, social security numbers, payroll information, health related information, and so on, is always secure. To achieve this, a strong data encryption solution is indispensable.

In DB2 for i versions previous to version 7.1, column triggers were utilized to automate encryption. While non-SQL native interfaces weren't able to automatically decrypt data, SQL interfaces could use an SQL view semi-transparently for the decryption, but this method required a lot of effort and was inconvenient.

The DB2 for i 7.1 release provides an enabling technology known as a field procedure (FieldProc) to deliver transparent column-level encryption implementations. By using the FIELDPROC clause of CREATE TABLE and ALTER TABLE, you can register a field procedure to a column. A field procedure is a user-written exit routine designed to transform values in a single column. When values in the column are changed, or new values are inserted, the field procedure is invoked for each value and can transform that value (encode it) in any way, and then store it. When values are retrieved from the column, the field procedure is invoked for each encoded value, which then decodes them back to the original value. Any indexes defined on a non-derived column that uses a field procedure are built with encoded values.
Advantages of field procedure support

You will benefit from the newly introduced field procedure encryption support in DB2 for i 7.1 in the following ways:

- Secure encryption. Data, index, and journals stored on hard disks or tapes are a transformed (encrypted) version of the data. No one can get the decrypted data without the FieldProc program.
- Easier usage. To enable field procedure, you just need to write a FieldProc program (or call third party encryption/decryption functions) and then register it to the table.
- More flexibility. You can easily change to other encryption methods by adding, dropping, or moving to other FieldProc programs. Note that even though an encoded value can be longer than the field's defined value, it is flexible and can be changed with FieldProc programs. No change to the original table definition is needed.

Performance considerations of field procedure support

While field procedures allow you to transparently implement encryption and decryption without application changes, this flexibility does have a performance tradeoff. Registering field procedure programs adds overhead, similar to that of an external program call, to every interface that writes or reads values from that column. The operations performed by the field procedure program itself will also factor into the performance overhead. For example, encryption and decryption processing are notorious for the amount of CPU cycles that they demand. Thus, performance testing should be a critical part of your field procedure implementation.

How to use the FieldProc

As an application developer, you can simply use the FIELDPROC clause to register the field procedure to a table. In the following example, an employee table is created with sensitive data (salary) that is encrypted by a field procedure:

```
Create TABLE employee
   
    ID char(10),
    salary decimal(10,2) FIELDPROC FP.userpgm

```

Or an existing employee table can be altered to encrypt the sensitive data in column salary:

```
ALTER TABLE employee alter column salary set FIELDPROC FP.userpgm
//We can alter to add a FieldProc on both SQL table and Native table.
```

Of course, the field procedure can also be dropped, and you can use another field procedure to encrypt the sensitive data:

```
ALTER TABLE FP.userpgm alter column salary drop FIELDPROC
//drop the old field procedure
ALTER TABLE employee alter column salary set FIELDPROC FP.userpgm2
//then add the new one
```
Restrictions of supported column attributes

DB2 for i 7.1 supports the encryption of any data type with field procedures except ROWID, IDENTITY and ROW CHANGE TIMESTAMP.

Notes: Only one LOB column can be added with FieldProc in each table.

When field procedures are invoked

A field procedure that is specified for a column is invoked in three general situations:

- In the case of field-definition, when the CREATE TABLE or ALTER TABLE statement that names the procedure is executed. During this invocation, the procedure is expected to determine the validity of data type and attributes of the column, verify the literal list, and provide the field description of the column.
- In the case of field-encoding, when a column value is to be field-encoded. This occurs for any value that is inserted by an SQL INSERT or MERGE statement, by a native write, is changed by update, or the data is copied to a target column with a field procedure registered.
- In the case of field-decoding, when a stored value is to be field-decoded back into its original value. This occurs for any value that is retrieved by an SQL SELECT or FETCH statement, by a native read, or the data is copied to a target column without a field procedure registered.

How to write a field procedure program

Three code samples are provided in the next section to illustrate the typical FieldProc program implementation.

Before introducing the field procedure program, it is necessary to explain several key new terms listed below. For the detailed information please refer the SQL Programming Guide.

- External format: The format of the column's data that will be either received or returned from/to a program from the database engine.
- Internal format: The format of the column's data as it is stored internally by the database engine.
- Field-encoding: The transformation your field procedure performs on a value.
- Field-decoding: The same routine which is used to undo the transformation when values are retrieved.
- Field-definition: The field procedure is also invoked when the table is created or altered, to define the data type and attributes of an encoded value to DB2.
  Note: A user-defined data type can be a valid field if the data type has any of the allowed SQL data types. DB2 casts the value of the column to the source type before it passes it to the field procedure.
- Column value descriptor (CVD): During field-encoding, the CVD describes the value to be encoded, as well as the decoded value to be supplied by the field procedure. During field-definition, it describes the column as defined in the CREATE TABLE or ALTER TABLE statement.
- Field procedure parameter list: The field procedure parameter list communicates general information to a field procedure, tells what operation is to be done, and allows the field
procedure to signal errors. DB2 provides storage for all parameters that are passed to the field procedure, therefore, parameters are passed to the field procedure by address.

- The following 8 parameters are in the list:
  - Parameter 1: A small (2 byte) integer that describes the function to be performed. This parameter is input only, and supports the following values:
    - 0 - field-encoding
    - 4 - field-decoding
    - 8 - field definition
  - Parameter 2: This is a structure that defines the field procedure parameter value list (FPPVL). For function code 8, this parameter is input/output. For function code 0 and 4, this parameter contains the output of the function code 8 call. This parameter is input only.
  - Parameter 3: This is the decoded data attribute that is defined by the Column Value Descriptor (CVD). This is the column attributes that were specified at `CREATE TABLE` or `ALTER TABLE` time. This parameter is input only.
  - Parameter 4: This is the decoded data. The usage of this parameter is dependent on function code.
  - Parameter 5: This is the Internal Encoded Data attribute that is defined by the Field Value Descriptor (FVD).
  - Parameter 6: This is encoded data that is defined by the Field Value Descriptor (FVD). The usage of this parameter is dependent on function code.
  - Parameter 7: The SQLSTATE is character(5). This parameter is input/output. Notes: This parameter is passed in from the database set to 00000, and can be set by the field procedure as a result state for the field procedure. While normally the SQLSTATE is not set by the field procedure, it can be used to signal to the database as follows: 38xxx - the function code detected an error situation, resulting in a SQL error. Here xxx may be one of several possible strings. Warnings are not supported for field procedures.
  - Parameter 8: The message text area is VARCHAR(1000). This parameter is input/output. Notes: This argument is set by DB2 to the empty string before calling the field procedure. It is a VARCHAR(1000) value that can be used by the field procedure to send message text back when an SQLSTATE error is signaled by the field procedure. It is initialized by the database on input to the field procedure, and may be set by the field procedure with descriptive information. Message text is ignored by DB2 unless the SQLSTATE parameter is set by the field procedure. The message text is assumed to be in the job CCSID.

Listing 1. This sample shows how to use encryption/decryption programs, or third party encryption/decryption functions to do the encryption. The program encrypts and decrypts a 16 character string that only includes numbers.

```c
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <SQLFP.h>

void DESC(unsigned char *data,
          unsigned char *mkey, char ctag[2])
{
    ... //user created or third party encryption algorithm.
}
```

Encryption enhancements: Field procedure support in DB2 for i 7.1
main(int argc, void *argv[])
{
    short *funccode = argv[1];
    sqlfpFieldProcedureParameterList_T *optionalParms = argv[2];
    char *sqlState = argv[7];
    sqlfpOptionalParameterValuePairDescriptor_T *optionalParmPtr;
    char KEY[16]="0123456789ABCDEF";

    if (*funccode == 8) /* create time */
    {
        sqlfpParameterDescription_T *inDataType = argv[3];
        sqlfpParameterDescription_T *outDataType = argv[5];
        if (inDataType->sqlfpSqlType !=452 &&
            inDataType->sqlfpSqlType !=453 )  /* only support fixed length */
            memcpy(sqlState, "38002", 5);
    
        /* do something here to determine the result data type */
        /* ...... */
        /* in this example input and output types are exactly the same */
        /* so just copy */
        memcpy(outDataType, inDataType, sizeof(sqlfpParameterDescription_T));
    }
    else if (*funccode == 0) /* encode */
    {
        memcpy((char *)argv[6], (char *)argv[4], 16);
    DESC((char*)argv[6], KEY, "0");
    }
    else if (*funccode == 4) /* decode */
    {
        memcpy((char *)argv[4], (char *)argv[6], 16);
        memcpy((char *)argv[4], (char *)argv[6], 16);
        DESC((char *)argv[4], KEY, "1");
    }
    else /* unsupported option -- error */
        memcpy(sqlState, "38003", 5);
}

Listing 2. This sample shows how to use IBM i APIs to encrypt and decrypt. The program could be used to encrypt and decrypt a fixed length char, varchar or clob.

<!-- In this example, there are some points which need attention: -->
<!-- a. The input/output data type could be a fixed char or varchar or clob. -->
<!-- But the block size of the encryption should be 16x(multiple time of 16). -->
<!-- So, we should define enough space to store the encryption result. -->
<!-- b. Avoid using pad option. Pad option will add a count of the pad characters -->
<!-- in the end of original string. -->
<!-- It may conduct one more block of the encryption result and the -->
<!-- column does not have enough space to store. -->
<!-- c. For better performance, ACTGRP(*CALLER), TERASPACE(*YES) and STGMDL(*INHERIT) -->
<!-- were recommended when you compile your program. -->
<!-- d. More information about IBM i Cryptographic Services APIs, -->
<!-- please refer to the link of the IBM i infocenter below: -->
<!-- http://publib.boulder.ibm.com/infocenter/iseries/7.1m0/index.jsp -->
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <QC3CCI.H>
#include <QUSEC.H>
#include <QC3DTAEN.H>
#include <QC3DTADE.H>
#include <SQLFP.h>
Encryption enhancements: Field procedure support in DB2 for i

7.1

```c
// SQL data type define.  
#define SQL_CLOB_1 408
#define SQL_CLOB_2 409
// SQL data type VARCHAR  
#define SQL_VARCHAR_1 448
#define SQL_VARCHAR_2 449
// SQL data type CHAR  
#define SQL_CHAR_1 452
#define SQL_CHAR_2 453

typedef _Packed struct
{
  unsigned short int len;
  char data[512];
}T_VARCHAR;

typedef _Packed struct
{
  unsigned long len;
  char data[512];
}T_CLOB;

Qc3_Format_ALGD0200_T *ALGD0200;
Qc3_Format_KEYD0200_T *KEYD0200;
Qus_EC_t ERRCODE;

main(int argc, void *argv[])
{
  T_VARCHAR VarCharStr;
  T_CLOB ClobStr;
  short *funcode = argv[1];
  sqlfpFieldProcedureParameterList_T *optionalParms = argv[2];
  char *sqlstate = argv[7];
  sqlfpMessageText_T *msgtext = argv[8];
  sqlfpOptionalParameterValueDescriptor_T *optionalParmPtr;
  char Clear_Data[512];
  char Encrypted_Data[512];
  char Decrypted_Data[512];
  int InputDataLen;
  int EncryptedDataLen;
  int DecryptedDataLen;
  int RtnLen;
  int i;
  char Qc3_Any_CSP_Flag = Qc3_Any_CSP;
  ALGD0200 = (Qc3_Format_ALGD0200_T *)malloc(sizeof(Qc3_Format_ALGD0200_T));
  ALGD0200->Block_Cipher_Alg = Qc3_AES;
  ALGD0200->Block_Length = 16;
  ALGD0200->Mode = Qc3_ECB;
  ALGD0200->Pad_Option = Qc3_No_Pad;
  ALGD0200->Pad_Character = '\x00';
  ALGD0200->MAC_Length = 0;
  ALGD0200->Effective_Key_Size = 0;
  ALGD0200->Reserved = '\x00';
  memset(ALGD0200->Init_Vector, '\x00', 32);

  KEYD0200 = (Qc3_Format_KEYD0200_T *)malloc(sizeof(Qc3_Format_KEYD0200_T) +16);
  KEYD0200->Key_Type = Qc3_AES ;
  KEYD0200->Key_String_Len = 16;
  KEYD0200->Key_Format = Qc3_Bin_String;
  memcpy((char *)&KEYD0200->Reserved + 3, "0123456789ABCDEF", 16);
}
```c
if (*funcCode == 8) /* create time */
{
    sqlfpParameterDescription_T *inDataType = argv[3];
    sqlfpParameterDescription_T *outDataType = argv[5];
    /* do something here to determine the result data type */
    /* ..... */
    /* in this example input and output types are exactly the same */
    /* so just copy */
    memcpy(outDataType, inDataType, sizeof(sqlfpParameterDescription_T));
}
else if (*funcCode == 0) /* encode */
{
    sqlfpParameterDescription_T *inDataType = argv[3];
    InputDataLen = inDataType->sqlfpByteLength;
    if (inDataType->sqlfpSqlType == SQL_VARCHAR_1 ||
        inDataType->sqlfpSqlType == SQL_VARCHAR_2)
    {
        memcpy((char *)&VarCharStr, (char *)argv[4], InputDataLen+2);
        InputDataLen = VarCharStr.len;
        memcpy((char *)&Clear_Data, (char *)VarCharStr.data, InputDataLen);
        if (InputDataLen % 16 > 0 || InputDataLen == 0)
        {
            memset((char *)Clear_Data + InputDataLen, '\x00',
                    16 - InputDataLen % 16);
            InputDataLen = ((int)(InputDataLen / 16) + 1) * 16;
        }
    }
    else if (inDataType->sqlfpSqlType == SQL_CLOB_1 ||
              inDataType->sqlfpSqlType == SQL_CLOB_2)
    {
        memcpy((char *)&ClobStr, (char *)argv[4], InputDataLen+4);
        InputDataLen = ClobStr.len;
        memcpy((char *)&Clear_Data, (char *)ClobStr.data, InputDataLen);
        if (InputDataLen % 16 > 0 || InputDataLen == 0)
        {
            memset((char *)Clear_Data + InputDataLen, '\x00',
                    16 - InputDataLen % 16);
            InputDataLen = ((int)(InputDataLen / 16) + 1) * 16;
        }
    }
    else
    {
        memset((char *)&Clear_Data, (char *)argv[4], InputDataLen);
        memset(EncryptedData, '\x00', sizeof(Encrypted_Data));
        EncryptedDataLen = sizeof(Encrypted_Data);
        Qc3EncryptData((char *)Clear_Data,
                       &InputDataLen,
                       &Qc3_Data,
                       (char *)ALG0200,
                       Qc3_Alg_Blk_Cipher,
                       (char *)KEY0200,
                       Qc3_key_Parms,
                       &Qc3_Any_csp_Flag,
                       Encrypted_Data,
                       &EncryptedDataLen,
                       &RtnLen,
                       &ERRCODE);
        if (inDataType->sqlfpSqlType == SQL_VARCHAR_1 ||
            inDataType->sqlfpSqlType == SQL_VARCHAR_2)
        {
            VarCharStr.len = RtnLen;
            memcpy((char *)VarCharStr.data, (char *)Encrypted_Data, RtnLen);
            memcpy((char *)argv[6], (char *)&VarCharStr, RtnLen+2);
        }
        else if (inDataType->sqlfpSqlType == SQL_CLOB_1 ||
                  inDataType->sqlfpSqlType == SQL_CLOB_2)
```

{ 
  ClobStr.len = RtnLen;
  memcpy((char *)ClobStr.data, (char *)Encrypted_Data, RtnLen);
  memcpy((char *)argv[6], (char *)&ClobStr, RtnLen+4);
}
else
  memcpy((char *)argv[6], (char *)Encrypted_Data, RtnLen);
}
else if (*funccode == 4) /* decode */
{
  sqlfpParameterDescription_T *inDataType = argv[3];
  InputDataLen = inDataType->sqlfpByteLength;
  if (inDataType->sqlfpSqlType == SQL_VARCHAR_1 ||
      inDataType->sqlfpSqlType == SQL_VARCHAR_2)
  {
    memcpy((char *)&VarCharStr, (char *)argv[6], InputDataLen+2);
    InputDataLen = VarCharStr.len;
    memcpy((char *)Encrypted_Data, (char *)VarCharStr.data, InputDataLen);
  }
  else if (inDataType->sqlfpSqlType == SQL_CLOB_1 ||
            inDataType->sqlfpSqlType == SQL_CLOB_2)
  {
    memcpy((char *)&ClobStr, (char *)argv[6], InputDataLen+4);
    InputDataLen = ClobStr.len;
    memcpy((char *)Encrypted_Data, (char *)ClobStr.data, InputDataLen);
  }
  else
    memcpy((char *)Encrypted_Data, (char *)argv[6], InputDataLen);
  memset(Decrypted_Data, '\00', sizeof(Decrypted_Data));
  DecryptedDataLen = sizeof(Decrypted_Data);
  Qc3DecryptData(Encrypted_Data,
                &InputDataLen,
                (char *)ALGD0200,
                Qc3_Alg_Block_Cipher,
                (char *)KEYD0200,
                Qc3_Key_Parms,
                &Qc3_Any_CSP_Flag,
                "",
                Decrypted_Data,
                &DecryptedDataLen,
                &RtnLen,
                &ERRCODE);
  if (inDataType->sqlfpSqlType == SQL_VARCHAR_1 ||
      inDataType->sqlfpSqlType == SQL_VARCHAR_2)
  {
    VarCharStr.len = strlen(Decrypted_Data);
    memcpy((char *)VarCharStr.data, (char *)Decrypted_Data, VarCharStr.len);
    memcpy((char *)argv[4], (char *)&VarCharStr, VarCharStr.len+2);
  }
  else if (inDataType->sqlfpSqlType == SQL_CLOB_1 ||
            inDataType->sqlfpSqlType == SQL_CLOB_2)
  {
    ClobStr.len = strlen(Decrypted_Data);
    memcpy((char *)ClobStr.data, (char *)Decrypted_Data, ClobStr.len);
    memcpy((char *)argv[4], (char *)&ClobStr, ClobStr.len+4);
  }
  else
    memcpy((char *)argv[4], (char *)Decrypted_Data, RtnLen);
}
else /* unsupported option -- error */
  memcpy(sqlstate, &"38003",5);
}
Listing 3. This sample is an RPG version of code example for FieldProc. It reverses the characters in a lob or varchar column as the encryption method.

```rpg
D FuncCode S 2B 0
D p_FuncCode S *
D OptParms DS LikeDs(SQLFOPVD)
D* D EnCodTyp DS LikeDs(SQLFPD)
D* D DeCodTyp DS LikeDs(SQLFPD)
D* D EnCodDta S 512
D DeCodDta S 512
D* D SqlState S 5
D SqMsgTxt DS LikeDs(SQLFMT)
D* D En_Lob_Ds Ds Qualified
D Len 5B 0
D Data 1 dim(512)
D
D De_Lob_Ds Ds LikeDs(En_Lob_Ds)
D
D En_VChar_Ds Ds Qualified
D Len 2B 0
D Data 1 dim(512)
D
D De_VChar_Ds Ds LikeDs(En_VChar_Ds)
D
D i S 10I 0
D
/COPY QSYSINC/QRPGLESRC,SQLFP
C   *Entry     Plist
C     Parm     FuncCode
C     Parm     OptParms
C     Parm     DeCodTyp
C     Parm     DeCodDta
C     Parm     EnCodTyp
C     Parm     EnCodDta
C     Parm     SqlState
C     Parm     SqMsgTxt
C
/Free
If FuncCode = 8 ;
   // do something here to determine the result data type
   // ....
   // in this example input and output types are exactly the same
   // so just copy
   EnCodTyp = DeCodTyp ;
ElseIf FuncCode = 0 ; // encode
   If DeCodTyp.SQLFST = 408 or DeCodTyp.SQLFST = 409 ; // clob
      // in this example, reverse the characters as encryption
      De_Lob_Ds = DeCodDta ;
      En_Lob_Ds.Len = De_Lob_Ds.Len ;
      i = De_Lob_Ds.Len ;
      DoW i > 0 ;
         En_Lob_Ds.Data(De_Lob_Ds.Len-i+1) = De_Lob_Ds.Data(i) ;
         i = i - 1 ;
      EndDo ;
      EnCodDta = En_Lob_Ds ;
   ElseIf DeCodTyp.SQLFST = 448 or DeCodTyp.SQLFST = 449 ; // varchar
      // in this example, reverse the characters as encryption
      De_VChar_Ds = DeCodDta ;
      En_VChar_Ds.Len = De_VChar_Ds.Len ;
      i = De_VChar_Ds.Len ;
      DoW i > 0 ;
         En_VChar_Ds.Data(De_VChar_Ds.Len-i+1) = De_VChar_Ds.Data(i) ;
   EndIf
```

Encryption enhancements: Field procedure support in DB2 for i

7.1
Special considerations when using the FieldProc

There are several guidelines that you should take into consideration when adopting field procedure support. The key items include:

- The field procedure must be an ILE *.PGM object. Note that *.SRVPGMs, OPM *.PGMs, and JAVA objects are not supported.
- No SQL is allowed in a field procedure.
- The field procedure must be deterministic. It is very dangerous to use field procedures that are not deterministic because the encoded data might not be able to decoded back to its original value.
- When the Change Physical File (CHGPF) command is used with the SRCFILE parameter to change the field definition of a physical file, the CHGPF command will remove all registered Field Procedures on that physical file without any warning message.

For more restrictions, please refer to SQL reference for details.

Conclusion

This article has given a glimpse into the capabilities that are available through field procedure support in IBM i 7.1. With the utilization of field procedure support, you can easily perform a
transparent column-level encryption of the data stored in a database. You can use FieldProc for any type of encoding scheme - it is not limited to encryption.

There are quite a few encryption methods that you can choose. For example, FieldProc of AES, or triple DES, and others are commonly used. Of course, you can also choose a third party encryption product such as Linoma, Patrick Townsend, and nuBridges, as an easy and complete solution. But when you implement IBM i applications, the field procedure support is a strong data encryption solution.

Acknowledgement

Thanks very much to our Rochester experts who helped review the article. Special thanks to Kent Milligan who gave a lot of valuable comments.
Related topics

- **Field procedure description in SQL reference**: Descriptions in information center of how to define and use field procedures.
- Refer to the IBM i Database SQL programming guide for more information.
- Download the iSeries Access for Windows Service Packs.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)