Protecting i5/OS data with encryption

Kent Milligan and Beth Hagemeister
ISV Business Strategy and Enablement
October 2006

Table of contents

Abstract

This paper focuses on the encryption of data at rest (primarily, the data stored in IBM DB2 tables and physical files). Recommended practices for encryption-key management are also discussed.

Introduction

Reports of personal data being stolen or compromised are heard on a regular basis, so it is no surprise that consumers are demanding more in terms of data protection and privacy. Thus, there is strong demand for encryption of data on the IBM® System i™ platform and all other computers in the world.

Before pursuing new methods of data protection with encryption, it is important not to overlook the first step in securing your IBM DB2® data: object-level security. Utilization of the robust IBM i5/OS® (and IBM OS/400®) object-level security services is critical in ensuring that your DB2 data is protected, regardless of which interface is used to access the data. With users requiring more data-access interfaces, System i shops cannot rely on menu-based security to control access to sensitive data. Object-level security prevents unauthorized users from accessing or changing sensitive company financial data (for example, earnings data).

Data encryption is a security method that you can use to provide another layer of protection around DB2 columns that contain sensitive data. This extra level of security is needed because object-level security cannot prevent authorized users (such as help-desk personnel) from viewing sensitive data. Nor can it prevent a hacker from reading sensitive data using credentials (ID and password) stolen from an authorized user. If sensitive data such as a credit-card number is stored in an encrypted format, then by default all users are always returned a binary string of encrypted data. To view the actual credit-card number, the user must be authorized to access the DB2 object as well as have access to the encryption key and the decryption algorithm.

Cryptography overview

Encryption and decryption algorithms are collectively called “cryptographic systems.” Cryptography is the science of keeping data secure. Cryptography itself can be defined as the process of scrambling plain text (or clear text) into cipher text (in other words, encryption), then back into plain text (that is, decryption). Figure 1 contains a graphical representation of a simple cryptographic system.

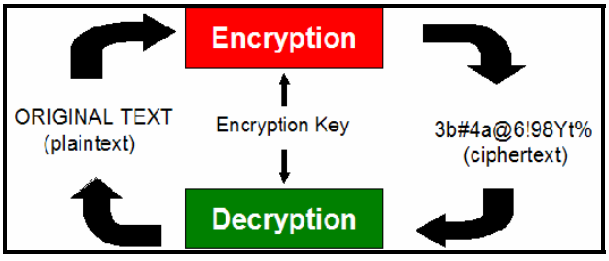

Figure 1. A simple cryptographic system

Before reviewing the cryptography support available with i5/OS, it is best to cover some of the basic operations associated with cryptography.

Hashing

Hashing is an important cryptographic operation that uses a mathematical algorithm to produce a hash (or message digest), as depicted in Figure 2.

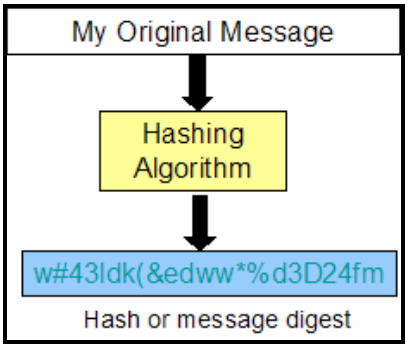

Figure 2. Hashing uses a mathematical algorithm

A cryptographic hash is called a one-way hash. For all practical purposes, the original clear text cannot be derived from the hash value because of the mathematical algorithm that is employed. A second important property of a cryptographic hash is that it is collision-resistant. That is, it is next to impossible to find another set of data that produces the same hash value. These two properties make cryptographic hashes useful for authentication purposes.

Some of the available hashing algorithms include Message Digest 5 (MD5), Secure Hash Algorithm (SHA)-1 SHA-256, SHA-384 and SHA-512. Protocols, such as SSL and IPSec, widely use a variant of SHA. Weaknesses have been found in the MD5 and SHA-1 algorithms and should not be used except for compatibility purposes.

From a data-security perspective, hashing is primarily used to protect the integrity of a data value. For instance, you can keep a copy of a digest for the purpose of comparing it with a newly generated digest at a later date. If the digests are identical, the data has not been altered. Also, hash algorithms are used in the generation of digital signatures.

Cryptographic hash algorithms are often used in another operation called a keyed-hash message authentication code (HMAC). An HMAC operation uses a secret key and a hash algorithm multiple times to produce a value (the HMAC) that later can be used to ensure that the data has not been modified. Typically, an HMAC is appended to the end of a transmitted message. The receiver of the message uses the same HMAC key and algorithm as the sender to reproduce the HMAC. If the receiver's HMAC matches the HMAC sent with the message, the data has not been altered.

A hashing algorithm is viewed as one-way encryption because the original value cannot be derived from the hash value. In contrast, encryption algorithms are used to make data unreadable and protect the confidentiality of data. Only those users with access to the encryption algorithm and key can access the original data.

Symmetric encryption

Symmetric cryptography uses the same key for both the encryption and decryption of the plain text (see Figure 3). This is the most popular method for encrypting data.

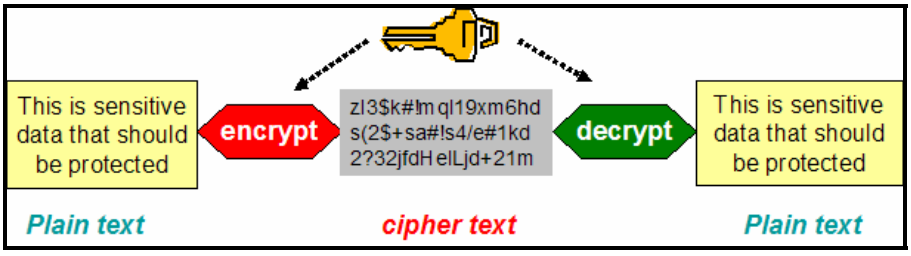

Figure 3. Most popular encryption data method

The most commonly used symmetric algorithms include: Data Encryption Standard (DES), Triple DES (3DES), Advanced Encryption Standard (AES), Rivest's Cipher (RC)2 and RC4. However, DES is no longer considered secure and should only be used for compatibility purposes.

Asymmetric encryption

With asymmetric cryptography, a pair of keys is used to encrypt and decrypt data. Data encrypted with the first key can only be decrypted with the second key, as demonstrated in Figure 4.

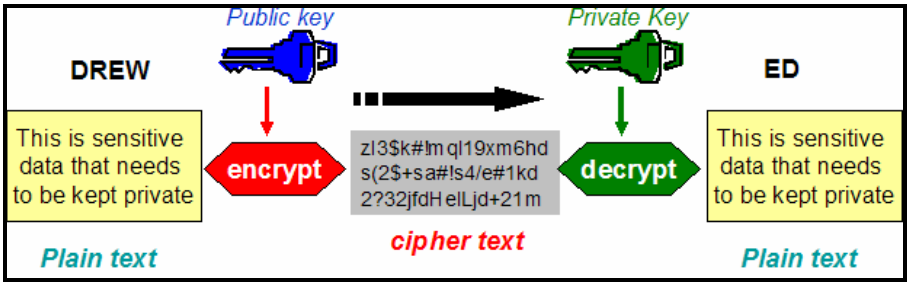

Figure 4. Asymmetric encryption

The two keys are mathematically related. Asymmetric encryption is more resource-intensive than symmetric encryption; that is why this method is not used as often for data encryption. Asymmetric encryption is often used in authentication, such as for the “handshake” performed by VPNs and SSL, and key distribution.

Encryption keys

As you saw in the earlier examples, encryption requires the use of encryption keys. Management and protection of these encryption keys is obviously a very important factor in keeping encrypted data safe. You probably have experienced this first-hand with the scrutiny given to creating nontrivial passwords and regularly changing them to secure access to the e-mail and other applications on your office systems.

Many of the same policies need to be applied to encryption keys. In addition, a decision is required on how to store and retrieve the encryption keys securely for those applications that need to encrypt and decrypt data. When the encryption keys are stored on the system, a backup plan must also be implemented to allow for recovery from a server failure or migration of the applications to another system. Recommended practices for key management are discussed later in this paper.

i5/OS and OS/400 encryption services

Both hardware and software encryption capabilities are available on i5/OS. As of i5/OS V5R4, no solutions automatically encrypt “data at rest” and automatically decrypt the data as it is retrieved from DB2 tables. This is primarily because this encryption approach only provides minimal data protection. Yes, the data is protected if someone steals the physical disk drive from the server. However, automatic decryption of encrypted data does allow any authorized user to see the sensitive data. Encryption and decryption must be performed explicitly by the application.

Here is a summary of the software-based encryption services that are available on i5/OS from IBM:

- Cryptographic services APIs
- Common Cryptographic Architecture (CCA) APIs
- Cryptographic Support for AS/400 (Cryptographic APIs) (support to be withdrawn after i5/OS V5R4)
- DB2 built-in SQL encryption and decryption functions
- CIPHER Machine Instruction (MI)
- IBM Java™ Cryptography Extension (IBMJCE) and IBM Java Cryptography Extension (IBMJCEFIPS) support

A couple of cryptographic hardware options are also available. These hardware features can offload the main CPU by performing cryptographic functions and increase security by providing a secure key store. The cryptographic hardware obtainable for i5/OS is:

- Cryptographic Accelerator 2058
- IBM Cryptographic Coprocessor 4758
- PCI-X Cryptographic Coprocessor 4764

Note: All new development must be done with the 4764 cryptographic coprocessor because IBM plans to withdraw support for the 2058 and 4758 in a future release.

i5/OS encryption interfaces

The following sections contain more-detailed information on each of the encryption interfaces.

CIPHER MI

The IBM System i cryptographic capabilities date back to the IBM System/38™ in the late 1970s.

The Cryptographic Support licensed program (LP) (57*xx*-CR1) included easy-to-use APIs and command language (CL) that provided DES encrypt, decrypt and message-authentication code (MAC) functions.

The CIPHER MI was the encryption interface for providing support for DES when IBM started shipping the IBM AS/400® in 1998. The repertoire of encryption interfaces has been expanded with the following cryptographic support:

- Encryption algorithms: DES, Triple DES, RC4-compatible, AES
- Hashing algorithm : SHA-1, MD5
- UNIX® crypt(3) function
- Pseudo-random number generation (PRNG) algorithms

Users of this MI interface are responsible for their own key management. Because of its longevity, CIPHER MI is the only encryption interface available in the base OS/400 and i5/OS releases prior to i5/OS V5R4.

Using CIPHER MI is not recommended for new applications because this support might not be enhanced with new algorithms or performance improvements in future i5/OS releases. Details on the CIPHER instruction can be found in “Machine Interface Instructions” under “APIs” by category in the IBM eServer™ iSeries™ Information Center.

CCA APIs

Over the years, industry standards changed dramatically. To meet higher security demands, the System i platform added support for hardware encryption, first with the IBM 2620 Cryptographic Coprocessor (which is no longer supported), then with the IBM 4758 Cryptographic Coprocessor, and most recently with the IBM 4764 Cryptographic Coprocessor. All of these support a much richer and more-secure API set than 57*xx*-CR1. This API set was known as the Common Cryptographic Architecture (CCA) APIs.

The CCA API is designed so that a call can be issued from essentially any high-level programming language. The call, or request, is forwarded to the cryptographic-services access layer and receives a synchronous response; that is, your application program loses control until the access layer returns a response after processing your request. CCA APIs are used together with the IBM 4758 or 4764 Cryptographic Coprocessor.

The CCA APIs require the installation of i5/OS 5722-SS1 option 35 (CCA Cryptographic Service Provider [CCA CSP]). CCA CSP provides API support equivalent to the CCA Support Program that is available for the IBM System z™, IBM System p™ and IBM System x™ platforms.

The Common Cryptographic Architecture (CCA) APIs provide a variety of cryptographic processes and data-security techniques. Applications using this API set can perform the following functions:

- Encrypt and decrypt information, typically using the 3DES algorithm in the cipher-block-chaining (CBC) mode to enable data confidentiality
- Hash data to obtain a digest, or process the data to obtain a MAC
- Create and validate digital signatures to demonstrate both data integrity and form the basis for nonrepudiation
- Generate, encrypt, translate and verify finance-industry PINs and transaction-validation codes with a comprehensive set of finance-industry-specific services
- Manage the various DES and Rivest-Shamir-Adleman algorithm (RSA) keys necessary to perform the above operations
- Control the initialization and operation of CCA

The *CCA Basic Services Reference and Guide* (in the IBM eServer iSeries Information Center) provides details on this API set.

Cryptographic Services APIs

In OS/400 V5R2, a new infrastructure was implemented within the Licensed Internal Code (LIC) that provided a common interface for LIC components to the various cryptographic-service providers (that is, the various pieces of hardware and software that implement cryptographic algorithms). In i5/OS

V5R3, the Cryptographic Services APIs were implemented on top of that infrastructure. These APIs provide applications with access to cryptographic services within the LIC or on the 2058 Cryptographic Accelerator. Only some of the cryptographic services support the 2058 device. A subset of these APIs is available on OS/400 V5R2 (through PTF SI10060, SI10105 and MF31101).

To use the Cryptographic Services APIs on an OS/400 V5R2 or i5/OS V5R3 system, the Cryptographic Access Provider product (57xx-AC3) must be installed. With the introduction of i5/OS V5R4, the full cryptographic support is enabled by default. Older versions of the Cryptographic Access Provider product (57xx-AC1 and 57xx-AC2) became obsolete and were dropped after government export restrictions on cryptographic function were reduced.

The i5/OS Cryptographic Services APIs help you to extend your application in the following ways:

- Privacy of data
- Integrity of data
- Authentication of communicating parties
- Nonrepudiation of messages

The Cryptographic Services API set was designed to be easier to use than the CCA APIs in high-level language programs. The APIs themselves can be broken into six categories, as follows:

- Encryption and Decryption APIs
- Authentication APIs
- Key Generation APIs
- PRNG APIs
- Cryptographic Context APIs
- Key Management APIs

A detailed discussion and examples of using these APIs are found later in this paper. In addition, the IBM eServer iSeries Information Center contains both C and RPG programming examples. The Cryptographic Services APIs provide the following cryptographic capabilities:

- Encryption algorithms: DES, 3DES, RC4-compatible, RC2, RSA and AES
- Hashing algorithms: MD5, SHA-1, SHA-256, SHA-384, SHA-512
- Key Exchange Algorithms: Diffie-Hellman
- Pseudo random number and key generation algorithms (PRNG)

SQL encrypt and decrypt functions

In i5/OS V5R3, built-in functions were added to allow data to be encrypted and decrypted on SQL requests. The SQL encryption and decryption functions actually use the Cryptographic Services APIs to perform the operations. These SQL functions provide a much simpler interface for encrypting and decrypting data, but they lack the overall capabilities of the Cryptographic Services APIs.

Here is a simplified example of the SQL cryptographic support in action. The encryption key is specified with the SET ENCRYPTION PASSWORD SQL statement.

```
SET ENCRYPTION PASSWORD = <character variable>
INSERT INTO emp VALUES(ENCRYPT_RC2('112233'), 'BOB SANDERS' )
SELECT DECRYPT_CHAR(id), name FROM emp
```

The ENCRPYT_RC2 and DECRYPT_RC2 functions then use this key to perform the respective encryption and decryption operations. The RC2 block-cipher-encryption algorithm is the only algorithm supported by DB2 for i5/OS in i5/OS V5R3. The encryption key is derived by hashing the specified password with the MD5 hashing algorithm. In i5/OS V5R4, support was added for the TDES encryption algorithm with the ENCRYPT_TDES and DECRYPT_TDES functions. In addition, the password used as input to these functions is derived using the SHA-1 hash algorithm.

Although these functions can only be run on SQL-based interfaces, some workarounds are available for non-SQL (that is, native) interfaces. These circumventions, along with more details on the SQL cryptography support, are covered in a later section of this paper.

Java-based interfaces

The IBM Java-based cryptography interfaces, IBM JCE and IBM JCEFIPS, are supported by System i models along with the other IBM systems. AES, DES and TDES are some of the encryption methods supported by the IBM Java Cryptography Extension Provider. The IBM JCE also includes support for hashing algorithms and a random number generator.

The IBM JCE is a standard extension to the Java 2 Software Development Kit (J2SDK), Standard Edition. The JCE implementation on i5/OS is compatible with the Sun Microsystems implementation. The IBM eServer iSeries Information Center contains documentation regarding the unique aspects of the i5/OS implementation. A listing for this site is provided in Appendix B.

The IBMJCE and IBMJCEFIPS do not use any of the i5/OS cryptography interfaces. All of the cryptography support is completely implemented in Java.

Key management

The safety of encrypted data depends on the security of your keys. This is no different than the physical security of your home or business. Even if the door locks can be the most sophisticated and expensive that money can buy, if your keys are easily accessible, the locks are useless.

Here is a list of the questions to address when protecting and managing encryption keys:

- Where is the encryption key stored?
- How are encryption keys generated?
- How is the encryption key protected?
- How is the encryption key retrieved and used by applications?
- How often will the encryption key change?
- Who manages encryption keys?
- What is the backup strategy for encryption keys?

Ideally, the keys must be managed and used by different personnel within your company. There is no reason for the application programmer to have direct access to the keys; instead, an interface can be provided to allow only programmatic retrieval of the key value. When identifying the people responsible for key management, it is also a good time to review the number of user profiles that have *ALLOBJ (all object) authority and restrict these users by putting additional controls in place, such as requiring the use of programs to decrypt sensitive data or keys. These programs can increase security by explicitly excluding *ALLOBJ authority.

How often you change cryptographic keys is another important consideration in your key-management strategy. In some cases, however, decrypting and encrypting the sensitive data with a new key might put the sensitive data more at risk than allowing the data to remain encrypted with an old encryption key. Thus, the frequency of changing the cryptographic keys will probably be different in each environment.

As mentioned earlier, a secure design should only allow programmers to access the encryption key using a programmatic interface. Given that fact, where does this programming interface retrieve the encryption key from? Some regulations or companies require that the keys be stored on a hardware device. The IBM Cryptographic hardware options (4764 and 4758, which were discussed earlier) support this capability.

In the next couple of sections, you will see that writing code to use i5/OS interfaces for encrypting and decrypting data is a simple matter of programming. The real challenge lies in designing a process and application architecture that selectively distribute the encryption keys to data consumers (users and programs) that are authorized to view the unencrypted version of the data. Application developers and security administrators must work together to design the best process for your application and business.

One alternative discussed earlier is having a single application that controls the decryption of the data. This program can be written to centralize the distribution of encryption-key passwords. A single program makes it easier to control which users are given the privilege to view the unencrypted data. Each time that data consumers need access to decrypted data, they would all use the same program. As that program shares the encryption password, it can also keep a log of the user requesting access and the timestamp of that activity for audit purposes. To limit the users that have authority to the centralized decryption program, it is wise to use program-adopted authority for the invoking programs and to exclude *ALLOBJ authority.

If hardware storage of the encryption keys is not required, then what other options exist on i5/OS? There are solutions from other software companies if you do not have the time or staff to implement your own solution. If interested in a solution from a software vendor, the companies that are part of the IBM System i Tools Innovation program are a good place to start (https://www-304.ibm.com/jct09002c/partnerworld/wps/pub/systems/i/technical/iii/tools?gcLang=en). If writing your own solution, then hardcoding the encryption password in the source code of a program or storing it in a data area is not a safe approach.

A user space is harder to access, but the encryption password is still being stored in the clear. One i5/OS object that provides more protection is a validation-list object. The list entry for a validation-list object consists of three parts: an ID, a data value that is encrypted and attributes for that entry. The fact that i5/OS encrypts the data in each list entry provides the additional protection. The person responsible for encryption keys uses the Add Validation List Entry or Change Validation List Entry operating system APIs to populate the validation-list object with encryption keys. Most likely, the entry ID is assigned a label that identifies that encryption key for a particular application or type of data. The application program passes in the appropriate label (entry ID) to retrieve the encryption key for encryption or decryption operations with the Find Validation List Entry API. The eServer iSeries Information Center provides more details on the programming APIs that are available for validation list objects. If a validation list object is used, make sure that *PUBLIC authority to the list object is revoked.

As shown with validation lists, data security usually improves with encryption of the data key itself. The Cryptographic Services technology enables this design approach with support for Key Encrypting Key (KEK), where one KEK is often used to protect a group of data-encryption keys. For example, one KEK is

encrypts several data-encryption keys that are established for protecting credit-card data; a second KEK is used to safeguard data-encryption keys that are associated with sensitive customer attributes (such as a Social Security number). The i5/OS Cryptographic Services provide yet another layer of security by encrypting the KEKs with a master key. These layers of protection are portrayed in Figure 5.

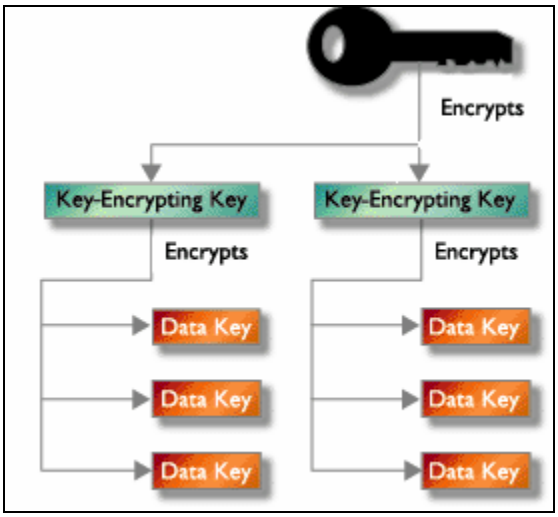

Figure 5. Master key layers of protection

In i5/OS V5R4, IBM introduced a set of new key-management APIs. These APIs allow you to create up to eight master keys and store them securely in the LIC. When a system is restarted (IPL), the master keys are available and all applications are operable without manual intervention. The next section of this paper covers this i5/OS V5R4 support in detail.

Wherever you decide to store your encryption keys, you need to make sure that there is a backup strategy for the keys. If you have a system failure and your recovery plans only cover your business data and not the encryption keys, then any decrypted data will be inaccessible without the keys that were used to encrypt it. Another critical success factor in your recovery plan is ensuring that the encryption keys are stored on different media than the encrypted data. If they are contained on the same media and that media is stolen or lost, then it is going to be much easier for your encrypted data to be compromised.

Cryptographic services

The Cryptographic Services APIs provide i5/OS application programmers with a powerful cryptographic tool set that they can use to help protect data privacy, ensure data integrity and authenticate communicating parties. The Cryptographic Services APIs are part of the base operating system and perform cryptographic functions within the i5/OS LIC or, optionally, on a 2058 Cryptographic Accelerator.

To help you get started in the use of Cryptographic Services APIs, this section:

- Provides a functional overview.
- Explains the Encrypt Data API parameters.
- Presents a simple encryption-example program.
- Discusses the use of algorithm contexts for extending operations over multiple calls.
- Reviews the key contexts for improving performance and protecting keys.
- Looks at key management APIs to help in the handling of cryptographic keys.

Functional overview

As mentioned earlier, the Cryptographic Services APIs break down into six categories of APIs:

- *Encryption and decryption APIs* include encrypt, decrypt and translate operations and are used for data confidentiality.
- *Authentication APIs* help you to ensure data integrity and authenticate the sender. They include hash, HMAC, MAC and signature operations.
- *Key generation APIs* generate both symmetric and asymmetric keys. They also include APIs for performing Diffie-Hellman key exchange (not supported on OS/400 V5R2).
- *PRNG APIs* generate cryptographically-secure pseudo-random values.
- *Cryptographic-context APIs* temporarily store key and algorithm information during cryptographic operations (not supported on OS/400 V5R2).
- *Key management APIs* help you store and handle cryptographic keys (introduced in i5/OS V5R4).

Figure 6 shows a list of the Cryptographic Services APIs by name and associated category.

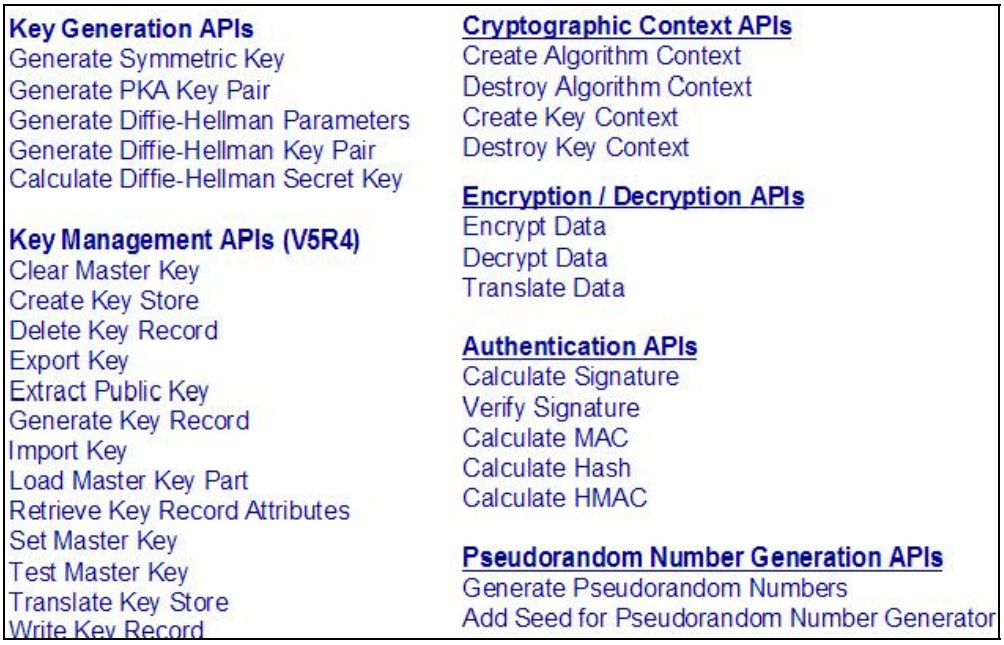

Figure 6. Cryptographic services APIs

Figure 7 displays the algorithms and key lengths included with the i5/OS V5R3 Cryptographic Services APIs. The i5/OS operating system contains all of these algorithms, and the 2058 Cryptographic Accelerator includes a subset (for a list of algorithms available on the 2058 Cryptographic Accelerator, see the article “OS/400 and 2058 Cryptographic Function Comparison” listed in Appendix B).

Algorithm	Key length or Modulus (in bits)
DES	56
Triple DES	56, 112, 168
AES	128, 192, 256
RC2	8–1024
RC4-compatible	8–2048
RSA	512–2048
MD5 Hash and HMAC	HMAC — a minimum of 128
SHA-1 Hash and HMAC	HMAC — a minimum of 160
SHA-256 Hash	N/A
SHA-384 Hash	N/A
SHA-512 Hash	N/A
Diffie-Hellman	512–1024
FIPS 186-1 pseudorandom number generation	N/A

Figure 7. Algorithms and key lengths

API parameters

At first glance, the Cryptographic Services APIs might look complex. They are designed to accommodate both simple operations, such as encrypting a single block of data with a clear key, as well as more-involved operations, such as encrypting data in multiple blocks during multiple calls with an encrypted key on a specific piece of hardware. However, the APIs are not as complicated as they appear.

To examine the Encrypt Data API, look at the C function prototype shown in Figure 8. Encrypt Data C function prototype (**Note:** The RPG prototype definitions for API formats are available in the QRPGLESRC(QC3CCI) file member in library QSYSINC.)

```
void Qc3EncryptData (
         char * volatile      , /* Clear data                   */
         int  * volatile      , /* Length of clear data         */
         char * volatile      , /* Clear data format name       */
         char * volatile      , /* Algorithm description        */
         char * volatile      , /* Algorithm desc format name   */
         char * volatile      , /* Key description              */
         char * volatile      , /* Key desc format name         */
         char * volatile      , /* Crypto Service Provider      */
         char * volatile      , /* Crypto Device Name           */
         char * volatile      , /* Encrypted data               */
         int  * volatile      , /* Len of encrypted data area   */
         int  * volatile      , /* Len of enc data returned     */
         void * volatile     ); /* Error Code                   */
```

Figure 8. Encrypt Data C function prototype

It is possible to divide the parameters in this prototype into six groups:

Parameters that describe the input data: There are three parameters of this type, as follows:

1.	Clear data	Input	Char(*)
2.	Length of clear data	Input	Binary(4)
3.	Clear data format name	Input	Char(8)

You specify the data to be encrypted in parameter 1 (clear data) in one of two ways, as follows:

- *A single block of data* is the simplest method of specifying the input data.
- *An array of data pointers and lengths* is used to encrypt data that is not in contiguous storage.

Parameter 3 (clear data format name) specifies which format is used in the clear data parameter. DATA0100 indicates it is a single block of data, and DATA0200 says it is an array. (**Note:** See the Cryptographic Services API documentation for the structure of the DATA0200 array, as well as for other structures discussed in this paper. Listings for this documentation is provided in Appendix B.) Parameter 2 (length of clear data) specifies the length of the data in the clear data parameter. For format DATA0100, this is the length of the clear data. For format DATA0200, it is the number of entries in the array.

Parameters that describe the cryptographic algorithm: The next two parameters appear on many of the Cryptographic Services APIs, as follows.

4.	Algorithm description	Input	Char(*)
5.	Algorithm desc format name	Input	Char(8)

As the names suggest, they describe the cryptographic algorithm. The supported cryptographic algorithms fall into one of four classes, as follows:

- Block-cipher algorithms (DES, TDES, AES and RC2)
- Stream-cipher algorithms (RC4-compatible)
- Public key algorithms (PKA) (RSA)
- Hash algorithms (MD5, SHA-1, SHA-256, SHA-384 and SHA-512)

(**Note:** Although the APIs support additional algorithms, the algorithm description parameter is not used to describe them.)

Each of the above algorithm classes requires its own set of arguments, and each has at least one argument, referred to as the algorithm type (for example, AES). Some algorithm types, such as block ciphers, have several additional arguments (such as block length, mode and pad option). Each of these algorithm classes has a defined structure for specifying the algorithm arguments. Parameter 4 (algorithm description) points to one of these structures; parameter 5 (algorithm-description format name) identifies which structure is used. The format names are as follows:

- ALGD0200 for the block-cipher arguments
- ALGD0300 for the stream-cipher arguments
- ALGD0400 for the PKA arguments
- ALGD0500 for the hash arguments

Of course, on the Encrypt Data API, the ALGD0500 structure is not valid because you cannot encrypt with a hash algorithm. In addition, ALGD0100 is a special algorithm-description structure that references an algorithm context. You use an algorithm context to extend cryptographic operations over multiple calls. This paper discusses algorithm contexts in more detail in a later section.

Parameters that describe the cryptographic key: Similar to the algorithm, the key is described with a structure of arguments using two parameters as follows:

6.	Key description	Input	Char(*)
7.	Key description format name	Input	Char(8)

Parameter 6 (key description) points to the structure, and parameter 7 (key description format name) identifies which structure is used, as follows:

- The KEYD0200 structure specifies the key type (for example, TDES and RSA), key format (binary or basic encoding rules [BER] encoded), key length and the key value.
- The KEYD0100 structure references a key context, which is used when the key is encrypted or is used on multiple operations (more about key contexts later).

Additional key description formats were added in i5/OS V5R4. These formats are described later.

Parameters that identify the CSP: The following two parameters identify the CSP that is used to perform the encryption and decryption, as follows.

8.	Cryptographic service provider	Input	Char(*)
9.	Cryptographic device name	Input	Char(8)

The i5/OS operating system supports several CSPs (the hardware or software that implements a set of cryptographic algorithms), such as the 2058 Cryptographic Accelerator, the 4758 Cryptographic Coprocessor, the i5/OS LIC and Java Cryptography Extensions. The Cryptographic Services APIs support two CSPs: the LIC and the 2058 Cryptographic Accelerator.

Parameter 8 (cryptographic service provider) specifies the CSP to perform the encryption. A value of 1 runs the encryption algorithm in the LIC, and a value of 2 runs the encryption algorithm on the first available 2058 Cryptographic Accelerator. If you desire a specific 2058 device, you can specify the device name in parameter 9 (cryptographic device name). If the 2058 Cryptographic Accelerator does not support the specified algorithm, an error is returned.

The easiest means of selecting a CSP is to specify a value of 0, which instructs the system to choose an appropriate CSP. The 2058 Cryptographic Accelerator is used if one is available and it supports the specified algorithm. Otherwise, the encryption is performed in the LIC. (**Note:** Some functions are always performed in the LIC. For a list of these functions, see “OS/400 and 2058 Cryptographic Function Comparison” listed in Appendix B.)

Parameters for the output data: Here are the three parameters in this category.

10.	Encrypted data	Output	Char(*)
11.	Length of area provided for encrypted dataChar(*)	Input	Binary(4)
12.	Length of encrypted data returned	Output	Binary(4)

Parameter 10 (encrypted data) and parameter 12 (length of encrypted data returned), both of which are output parameters, are where such data is returned. Parameter 11 (length of area provided for encrypted data) is an input parameter where you specify such information. Depending on the algorithm, the encrypted data can be larger than the clear data. For example, the length of RSA-encrypted data always equals the key size. Alternatively, if you specify padding on a block cipher, the last block of the clear data is padded to a full-block size before encryption. If the length of area provided for the encrypted data is too small, an error is returned without any encrypted data.

Parameter for the error code: Parameter 13, the final parameter, returns error information, as follows:

13. Error Code I/O Char(*)

The error code parameter is a variable-length structure that is common to all system APIs. (You can find documentation on the error code parameter at http://publib.boulder.ibm.com/infocenter/iseries/v5r4/ic2924/info/apis/error.htm#hdrerrcod.)

Many of the Cryptographic Services APIs use identical or similar parameters and structures to the ones just described.

A simple Encrypt Data API example

Figure 9 shows a simple example C program that uses the Encrypt Data API (Qc3EncryptData) to implement an AES Encrypt service program. The C header file for Qc3EncryptData is qc3dtaen.h. It contains the function prototype shown in Figure 8.

As you can see, all the parameters are passed as pointers. So, for example, when the API documents a parameter as Binary(4), what is actually specified on the call is a pointer to an int. Header file qc3dtaen.h pulls in another include, qc3cci.h, which contains the structures and constants required for using the Cryptographic Services APIs.

```
/*****************************************************************/
/* SRVPGM NAME:    AESEncrypt                                    */
/*                                                               */
/* DESCRIPTION: Perform AES encryption using                     */
/*                - CBC mode                                     */
/*                - 32-byte key                                  */
/*                - 16-byte block                                */
/*                - Pad with 0x00                                */
/*                                                               */
/* LANGUAGE:     ILEC                                            */
/* PARAMETERS:                                                   */
/*     ------------------------------------------------------    */
/*      PARM |  USE   | TYPE   | DESCRIPTION                     */
/*     ------------------------------------------------------    */
/*       1   | Input  | char * | Data to encrypt                 */
/*     ------------------------------------------------------    */
/*       2   | Input  | int    | Length of data to encrypt       */
/*     ------------------------------------------------------    */
/*       3   | Input  | char * | Key                             */
/*     ------------------------------------------------------    */
/*       4   | Input  | char * | Initialization vector           */
/*     ------------------------------------------------------    */
/*       5   | Output | char * | Place to put encrypted data     */
/*     ------------------------------------------------------    */
/*       6   | Input  | int  * | Size allocated for encrypted data */
/*           | Output |        | Actual length of encrypted data */
/*     ------------------------------------------------------    */
/*                                                               */
/* Use the following commands to compile this program:           */
/*   CRTCMOD   MODULE(my_lib/AESENCRYPT) SRCFILE(my_lib/my_src)  */
/*   CRTSRVPGM SRVPGM(my_lib/AESENCRYPT) MODULE(my_lib/AESENCRYPT) + */
/*             BNDSRVPGM(QC3DTAEN) EXPORT(*ALL)                  */
/*                                                               */
/*****************************************************************/

/*---------------------------------------------------------------*/
```

```
#include <string.h>                          /* C string utilities     */
#include                                     /* Error code structure   */
#include <qc3dtaen.h>                        /* qc3dtaen API header    */

int AESEncrypt(char * inputData, int inputLen, char * keyValue,
               char * iv, char * cipherData, int * cipherLen)
{
/*  Local variables                                                    */
Qc3_Format_ALGD0200_T   blockAlgD;           /* Algorithm description  */
struct {
  Qc3_Format_KEYD0200_T keyD;                /* Key description        */
  char                  keyS[256];           /* Key string             */
} key;

char                    csp;                 /* Crypto service provider*/
int                     rtnLen;              /* Return length          */
struct Qus_EC           errCode;             /* Error code parameter   */
/*  Start of executable code                                           */

/*  ------------SECTION A Start -----------------------------------*/
/* Set up the algorithm description                                   */
memset(&blockAlgD, 0, sizeof(blockAlgD));  /* Init to null           */
blockAlgD.Block_Cipher_Alg = Qc3_AES;      /* Algorithm is AES       */
blockAlgD.Block_Length = 16;               /* Block len is 16 bytes  */
blockAlgD.Mode = Qc3_CBC;                  /* Mode is CBC            */
blockAlgD.Pad_Option = Qc3_Pad_Char;       /* Pad with 0x00          */
memcpy(blockAlgD.Init_Vector, iv, 16);     /* Copy in init vector    */
/*  ------------SECTION A End   -----------------------------------*/

/*  ------------SECTION B Start -----------------------------------*/
/* Set up the key description                                         */
key.keyD.Key_Type = Qc3_AES;               /* Key type is AES        */
key.keyD.Key_Format = Qc3_Bin_String;      /* Key format is binary   */
key.keyD.Key_String_Len = 32;              /* Key len is 32 bytes    */
memcpy(key.keyS, keyValue, 32);            /* Copy in key value      */
/*  ------------SECTION B End   -----------------------------------*/

/*  ------------ SECTION C Start -----------------------------------*/
/* Set the cryptographic service provider                             */
csp = Qc3_Any_CSP;                         /* Use any CSP            */
/*  ------------ SECTION C End   -----------------------------------*/

/*  ------------ SECTION D Start -----------------------------------*/
/* Set up the error code parameter                                    */
memset(&errCode, 0, sizeof(errCode));
errCode.Bytes_Provided = 0;                /* Request exceptions     */
/*  ------------ SECTION D End   -----------------------------------*/

/*  ------------ SECTION E Start -----------------------------------*/
/* Encrypt the data                                                   */
Qc3EncryptData(inputData, &inputLen, Qc3_Data,
               (char*)&blockAlgD, Qc3_Alg_Block_Cipher,
               (char*)&key, Qc3_Key_Parms,
               &csp, NULL, cipherData, cipherLen, &rtnLen, &errCode);

*cipherLen = rtnLen;                       /* Return encrypted len   */
/*  ------------ SECTION E End -------------------------------------*/
return 0;
}                                          /* end AESEncrypt         */
```

Figure 9. Simple Encrypt Data API example

All the parameters are passed as pointers. For example, when the API documents a parameter as Binary(4), the call actually specifies a pointer to an int. Header file qc3dtaen.h pulls in another include, qc3cci.h, that contains the structures and constants required for using the Cryptographic Services APIs.

The following briefly explains the parameters on the call to Qc3EncryptData (see SECTION E, Figure 9).

- The **clear data** parameter and **length of clear data** parameter are passed into the service program. The **clear data** format name is set to constant `Qc3_Data` (at SECTION E). This constant is defined in `include qc3cci.h as DATA0100`.
- The **algorithm description** parameter contains the ALGD0200 structure. At SECTION A, the program sets up this structure for the AES algorithm with CBC mode, a block length of 16 bytes, no pad, and the initialization vector that was passed into the program. The **algorithm description format name** parameter is set to constant `Qc3_Alg_Block_Cipher` (at SECTION E). This constant is defined in `include qc3cci.h as ALGD0200`.
- The key description parameter contains the KEYD0200 structure. As you can see at SECTION B, it is set up for a 32-byte AES key with the key value that was passed into the program. The **key description format name** parameter is set to constant `Qc3_Key_Parms` (at SECTION E). This constant is defined in `include qc3cci.h as KEYD0200`.
- The cryptographic service provider parameter is set to constant `Qc3_Any_CSP` (at SECTION C). This constant is defined in `include qc3cci.h` with a value of `0`. A NULL pointer is passed for the **cryptographic device name** parameter (at SECTION E).
- The **encrypted data** parameter and length of area provided for it are passed into the service program. The **length of encrypted data returned** parameter is a local variable.
- The Bytes_Provided field of the **error code** parameter is set to `0` (at SECTION D), which will force errors to surface as exceptions.

Algorithm and key contexts

Algorithm and key contexts are temporary repositories for algorithm and key information. You create them with the Create Algorithm Context and Create Key Context APIs. These APIs each return an 8-byte token, which you can then specify on other cryptographic APIs. You use the Destroy Algorithm Context and Destroy Key Context APIs to destroy the algorithm and key contexts. Otherwise, they are destroyed at the end of the job.

***Algorithm contexts**:* Algorithm contexts store the algorithm type (for example, TDES and AES) and associated arguments (such as block size and pad character). They also store the state of a cryptographic operation. An algorithm context enables you to extend a cryptographic operation over multiple calls. For example, if you want to create an HMAC for an entire file, but you do not want to read the entire file before calling the Calculate HMAC API, you must first create an algorithm context. The following is the Create Algorithm Context API C function prototype.

```
void Qc3CreateAlgorithmContext (
       char * volatile        , /* Algorithm description        */
       char * volatile        , /* Algorithm desc format name   */
       char * volatile        , /* Algorithm context token      */
       void * volatile       ); /* Error code                   */
```

The API accepts four parameters as listed here.

1.	Algorithm description	Input	Char(*)
2.	Algorithm desc format name	Input	Char(8)
3.	Algorithm context token	Output	Char(8)
4.	Error code	I/O	Char(*)

As with the Encrypt Data API, you specify a structure containing the algorithm arguments in the **algorithm description** parameter. For an HMAC operation, the structure must contain the ALGD0500 arguments for a hash algorithm. Actually, there is only one argument, the type of hash algorithm (for example, SHA-1).

The parameters for the Calculate HMAC API look similar to those for the Encrypt Data API, as you can see from the C function prototype shown below.

```
void Qc3CalculateHMAC (
         char * volatile      , /* Input data                    */
         int  * volatile      , /* Length of input data          */
         char * volatile      , /* Input data format name       */
         char * volatile      , /* Algorithm description        */
         char * volatile      , /* Algorithm desc format name   */
         char * volatile      , /* Key description              */
         char * volatile      , /* Key desc format name         */
         char * volatile      , /* Crypto Service Provider      */
         char * volatile      , /* Crypto Device Name           */
         char * volatile      , /* HMAC                         */
         void * volatile     ); /* Error Code                   */
```

In this example, the **algorithm description format name** parameter must specify ALGD0100, meaning the **algorithm description** parameter is a structure that references an algorithm context. The following is the layout of the ALGD0100 structure:

```
Algorithm context flag     Char(8)
Final operation flag       Char(1)
```

You need to set the algorithm context token to the value returned on the Create Algorithm Context API. On each invocation of the Calculate HMAC API before the last invocation, set the final operation flag to 0 to keep the operation in a continued state. The algorithm context keeps a running HMAC and buffers any data that is not a full block size. On the last Calculate HMAC, set the final operation flag to 1; this value instructs the API to complete the operation and return the HMAC value.

You can use an algorithm context on different operations but only after completing each operation. In this example, after the final flag has been set to on and the HMAC value has been returned, you can reuse the algorithm context on a new operation, such as the Calculate Hash API or another Calculate HMAC.

***Key contexts**:* A key context holds key information, such as the key type (for example, MD5-HMAC, AES and RSA), key format (binary or BER-encoded) and key length. Key contexts serve several purposes:

- They improve the performance for some algorithms, such as AES and RC2. Some algorithms require initial key processing before performing the cryptographic operation. When you use a key context, initial key processing needs to be performed only one time.
- They allow the application to erase the **clear key** value from program storage to protect the key.

- They enable the application to use an encrypted key value with the APIs. You should reduce exposure of **clear key** values within the application program as much as possible. **Encrypted key** values can be used on an API if a key context is created. When you create a key context for an **encrypted key** value, you must supply the key and algorithm contexts for the KEK. Now, the issue becomes protection of the KEK. (For more on key management, see “Scenario: Key Management and File Encryption Using the Cryptographic Services APIs.” listed in Appendix B.)

i5/OS V5R4 enhancements

i5/OS V5R4 Cryptographic Services supports the specification of several new key formats. New structures for these formats have been defined for the **Key Description** parameter of the Encrypt Data, Decrypt Data, Calculate MAC, Calculate HMAC, Calculate Signature and Verify Signature APIs.

Key store label: With key description format KEYD0400, you can specify the file name, library and label for a key in a Cryptographic Services key store file on any of these APIs.

PKCS #5: With key description format KEYD0500, you can use RSA Security’s Public Key Cryptography Standard #5 to derive a symmetric key from a password, a salt and an iteration count. The password must be kept secret. The salt value, which does not need to be secret, is used to produce a large set of key possibilities for each password. Therefore, the salt should be a good random value. The iteration count is used to increase the length of computation. Using a large iteration value makes an exhaustive search for the key prohibitive. You can specify Password-Based Cryptography Standard (PKCS) #5 format on the Encrypt Data, Decrypt Data, Calculate MAC and Calculate HMAC APIs. (For more information about PKCS #5, refer to the standard listed in Appendix B.)

PEM formatted certificate: Privacy enhanced mail (PEM) is a standard for secure electronic mail over the Internet. A PEM-formatted certificate is a public-key certificate that is base64-encoded and has the text “-----BEGIN CERTIFICATE-----” and “-----END CERTIFICATE-----” appended to the beginning and the end. (A public-key certificate is basically a digital ID that can be verified. It contains a serial number, the owner’s name, the issuer’s name, validity dates, the public-key component of the owner’s RSA key pair and a digital signature that the issuer creates. Base64 is an encoding scheme in which any arbitrary sequence of bytes is converted into printable ASCII characters.) With key description format KEYD0600, you can specify a PEM-formatted certificate on the Encrypt Data, Decrypt Data and Verify Signature APIs. (For more information about PEM formatted certificates, refer to RFC 2127, listed in Appendix B.)

Certificate label: With key description format KEYD0700, you can specify a label that identifies a public key in a public-key certificate located in i5/OS Certificate Store. You can specify a certificate label on the Encrypt Data, Decrypt Data and Verify Signature APIs.

Distinguished name: With key description format KEYD0800, you can specify a distinguished name (the certificate owner) that identifies a public key in a public-key certificate located in i5/OS Certificate Store. You can specify a distinguished name on the Encrypt Data, Decrypt Data and Verify Signature APIs.

Application identifier: With key description format KEYD0900, you can specify an application identifier identifies the private key associated with a public-key certificate in i5/OS Certificate Store. You must use Digital Certificate Manager (DCM) to create an application definition and assign a certificate to it before performing an operation with a private key. You can specify an application identifier on the Encrypt Data, Decrypt Data and Calculate Signature APIs. For more information about DCM and i5/OS Certificate Store, refer to the IBM eServer iSeries Information Center article listed in Appendix B.

Cryptographic services and key management

As mentioned earlier, the level of protection provided by data encryption is dependent on how well the encryption keys are protected and managed. This section provides information on how to use Cryptographic Services for secure encryption keys.

Key generation

Having good random key values is extremely important. You can obtain good randomness by having the computer do this for you.

A computer function that generates a random value is called a pseudo-random number generator (PRNG). (The term “pseudo” is used because this function generally does not directly output from a real random source.) Many PRNGs exist; however, not all are cryptographically strong. “Cryptographically strong” means that the PRNG uses cryptographic functions to ensure that the output is unpredictably random. This is a much more-difficult feat than you might suppose. Many security systems have been broken because the PRNG was imperfect.

OS/400 V5R1 implemented a cryptographically strong PRNG. Many system components use the system PRNG. In addition, two APIs are available for your use: Generate Pseudo-random Numbers and Add Seed for Pseudo-random Number Generator.

Cryptographic Services has three APIs for generating key values: Generate Key Record (V5R4), Generate Symmetric Key and Generate PKA Key Pair. All three APIs use the system PRNG to generate cryptographically-strong pseudo-random key values.

The Generate Key Record API is new in i5/OS V5R4 and generates a random key or key pair and stores it in a key-store file. The key store file is discussed in more detail in the “i5/OS V5R4 Advancements in Key management” section of this paper.

i5/OS V5R3 considerations

On i5/OS V5R3, the Generate Symmetric Key and Generate PKA Key Pair APIs can be used to generate random key values for both symmetric and asymmetric algorithms.

The Generate Symmetric Key API generates a random key value for use with DES, Triple DES (TDES), AES, RC2 or RC4-compatible cryptographic algorithms. The Generate PKA Key Pair API generates a random RSA-key pair. You indicate the key type, key size and whether the key must be returned in clear data or encrypted format. If you specify an encrypted key, you must supply the key and algorithm contexts for the KEK.

The next diagram (see Figure 10) pictorially represents the process of using the Generate Symmetric Key API in combination with other cryptographic APIs to generate a set of three encryption key values: master key, key-encryption key and data-encryption key.

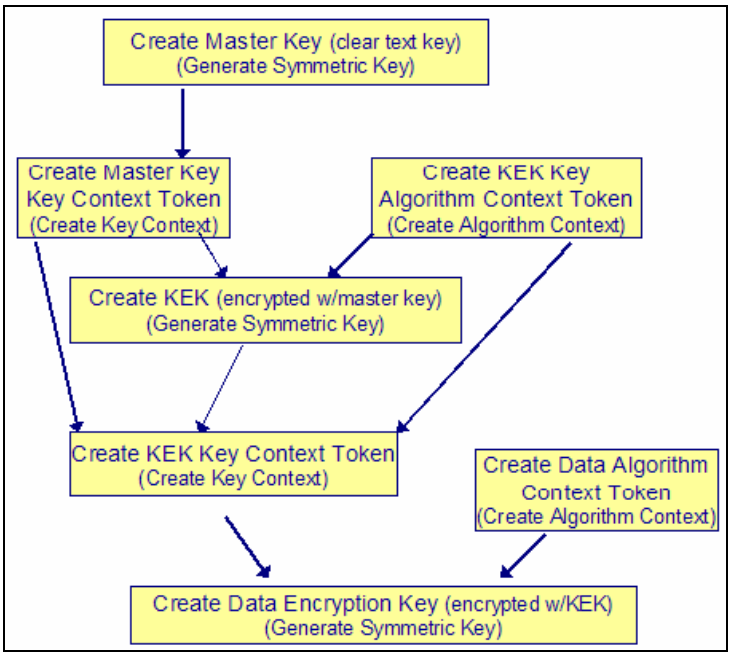

Figure 10. Generate Symmetric Key API

First, a master key is generated with the Generate Symmetric Key API. The result is a clear-text key. Because this key is in the clear, it must be protected by the application. This key is used as an input parameter for the Create Key Context API. This API creates a temporary area for holding a cryptographic key. An algorithmic context also needs to be created to describe a temporary area for holding the algorithm parameters and the state of the cryptographic operation. This API is called Create Algorithm Context. These APIs each return an 8-byte token for referencing the created context.

Next, the KEK is generated. This time, the Generate Symmetric Key API is also used, but the master key context token and the algorithm context token are used as input parameters. The generated key that the API provides in an output parameter is encrypted by the master key.

In the next step, you need to create another key context for the KEK. For the input parameters, you need to provide the encrypted KEK and the master key and algorithm context tokens (to decrypt the KEK).

Another algorithm context must be created to hold the algorithm information for encrypting the data-encryption key. You also need to provide the KEK key-context token. These two objects are used as input parameters for the last Generate Symmetric Key operation. This final operation generates the encrypted data-encryption key.

Key storage

If implementing an encryption solution on i5/OS V5R3, it is important to remember that the Cryptographic Services APIs provide no key management. Figuring out how to store and manage cryptographic keys is completely the application programmer's responsibility.

You must give careful consideration to authorities placed on objects containing keys. Even if the key is encrypted, users who have access to the KEK can hack the data. Encrypt keys stored on removable media and do not store the clear KEK with the encrypted keys.

Typically, KEKs are stored encrypted as well. You use a master key, which is the only key that is not encrypted, to encrypt KEKs. Ideally, you should not store the master key on the system. (For an example design for setting up and using a master key and KEKs, see “Scenario: Key Management and File Encryption Using the Cryptographic Services APIs” listed in Appendix B.)

Note that the 2058 Cryptographic Accelerator handles keys in the clear and does not provide key-storage facilities. If your application has a requirement to store and handle keys securely on tamper-resistant hardware, you might want to use the CCA API set, which performs cryptographic operations on the 4764 and 4758 Cryptographic Coprocessors (to learn more, see “Cryptographic Coprocessors for iSeries” listed in Appendix B).

i5/OS V5R4 advancements in key management

In i5/OS V5R4, Cryptographic Services adds new functionality to help with key management, including support for a hierarchical-key store.

Cryptographic-services key store

Figure 11 shows a diagram of a hierarchical-key system, a commonly used key-management technique.

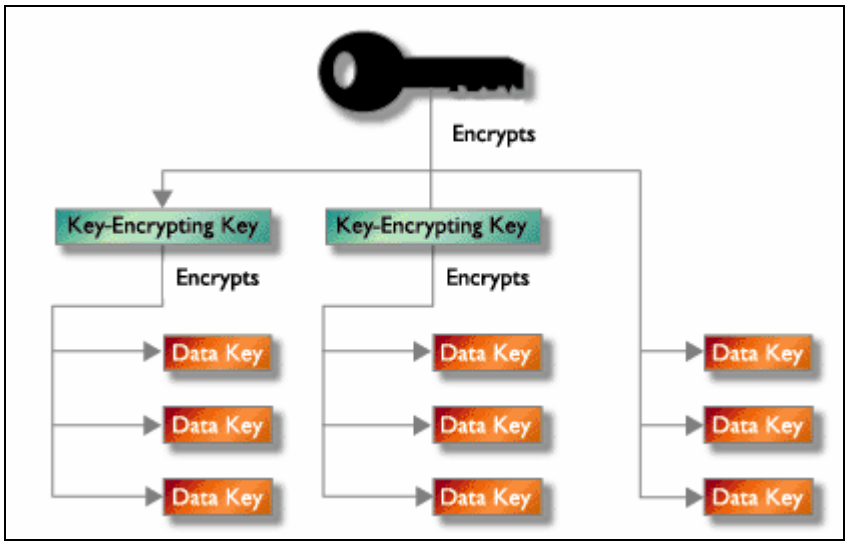

*Figure 11. The master key is at the top of the hierarchy and is a **clear key** value*

At the top of the hierarchy is a master key (or keys). The master key is the only *clear key* value and must be stored securely. You use KEKs to encrypt other keys. Typically, you use a KEK to encrypt a key that is sent to another system or stored outside the key store. KEKs are usually stored encrypted under a master key. Data keys are used directly on user data (for example, to encrypt or sign). You can encrypt a data key under a KEK or under a master key. The i5/OS Cryptographic Services implements such a key hierarchy through a two-tier key store.

Master keys

The i5/OS environment supports eight master keys that are used to encrypt other keys (that is, KEKs and data keys), but not data. Figure 12 represents how the master keys are stored in the i5/OS LIC in an area that cannot be displayed or dumped.

You can access the master keys only with the Cryptographic Services APIs. (For more information about Cryptographic Services master keys, refer to the eServer iSeries Information Center article listed in Appendix B.) To reference master keys, you simply use a number, 1 through 8.

Each master key is composed of three 32-byte values, called versions. The versions are new, current and old (see Figure 12). The new master key version contains the value of the master key while it is being loaded. The current master-key version contains the active master-key value. This is the value that will be used when a master key is specified on a cryptographic operation (unless specifically stated otherwise). The old master -key version contains the previous current master-key version. It is used to prevent the loss of data and keys when the master key is changed.

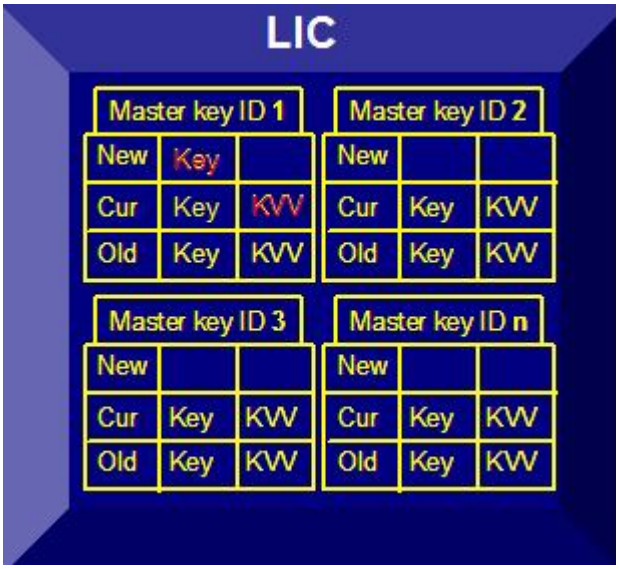

Figure 12. How master keys are stored in the LIC

To load and set (or activate) a master key, you use APIs. The Load Master Key API takes a passphrase as input. A passphrase is a sequence of text used to control access to a master key. You can load as many passphrases as desired before setting a master key. In fact, to ensure that no single individual has the ability to reproduce a master key, you need to assign passphrases to several people.

The Set Master Key API activates the new master-key value, which consists of the passphrases that were previously loaded. The API returns a key verification value (KVV). You need to retain this value; then at a later date, you can determine whether the master key has been changed. When you set a master key, any keys encrypted under that master key must be translated (that is, decrypted and then encrypted again).

A key hierarchy is a popular key-management technique primarily because it reduces the number of secrets (for example, plaintext keys and passwords) to protect. Be discriminating about how many master keys you actually need to use and must, therefore, manage.

Key-store files

You can securely store KEKs and data keys in a Cryptographic Services key-store file. A key-store file is a database file that you create with the Create Key Store API. Even though the key-store file is a database file, none of the normal data-access methods, such as SQL or record-level access, are able to access the key-store file. You specify the file name and library, the public authority and the master key that encrypts all the key values stored in the key-store file. Figure 13 shows a graphic representation of a key-store file.

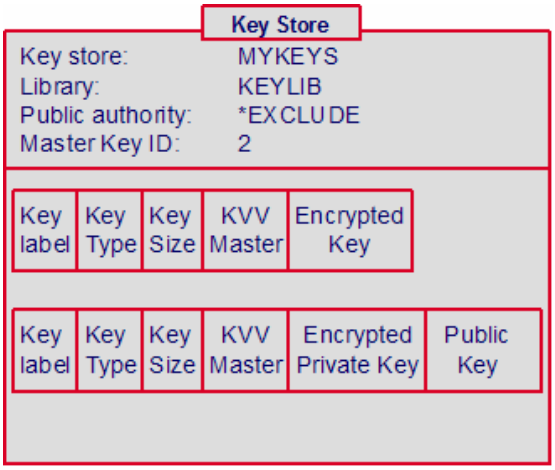

Figure 13. The key store

Both KEKs and data-encryption keys can be stored in the key-store file.

To store a key in a key-store file, use the Write Key Record API. Parameters let you specify:

- The type and size of the key
- Whether the input key value is a binary string, a BER-encoded string or a Privacy Enhanced Mail (PEM) certificate
- Whether the input key value is encrypted (and if so, the KEK information)
- A label for referencing the key

Alternatively, you can use the Generate Key Record API to generate and store a random key value in a key-store file, again specifying key type and size and a label for the key record.

To use a key from key store, you use a new structure defined for the Key Description parameter. This parameter is used by many of the Cryptographic Services APIs (for example, the Encrypt Data API) for specifying the key data. In this new structure, you specify the key-store file name, the library and the key-record label. Cryptographic Services retrieves the key value from the key-store file and decrypts it from under the master key before running the operation (for example, encrypting data).

Careful considerations must be given to authorities when creating key-store files. Even though key values in a key-store file are encrypted, anyone with access to the key-store file and the appropriate API (for example, the Decrypt Data API) can hack the data. (**Note:** A key-store file that is moved to another system is useless if the master key under which it was encrypted has not been set identically.)

Master-key encrypted keys

As stated earlier, key values in a key-store file are encrypted under a master key. You can also encrypt under a master key outside of a key store. On the Generate Symmetric Key API and the Generate PKA Key Pair API, you can request that the generated key value be returned encrypted under a master key. Or, use the Import Key API to encrypt a **clear key** value or translate (for example, decrypt, then encrypt) an encrypted key value under a master key. Again, you must carefully control access to master-key-encrypted keys.

Figure 14 demonstrates the process of the master key and key-store file being used to generate a KEK and data-encryption key.

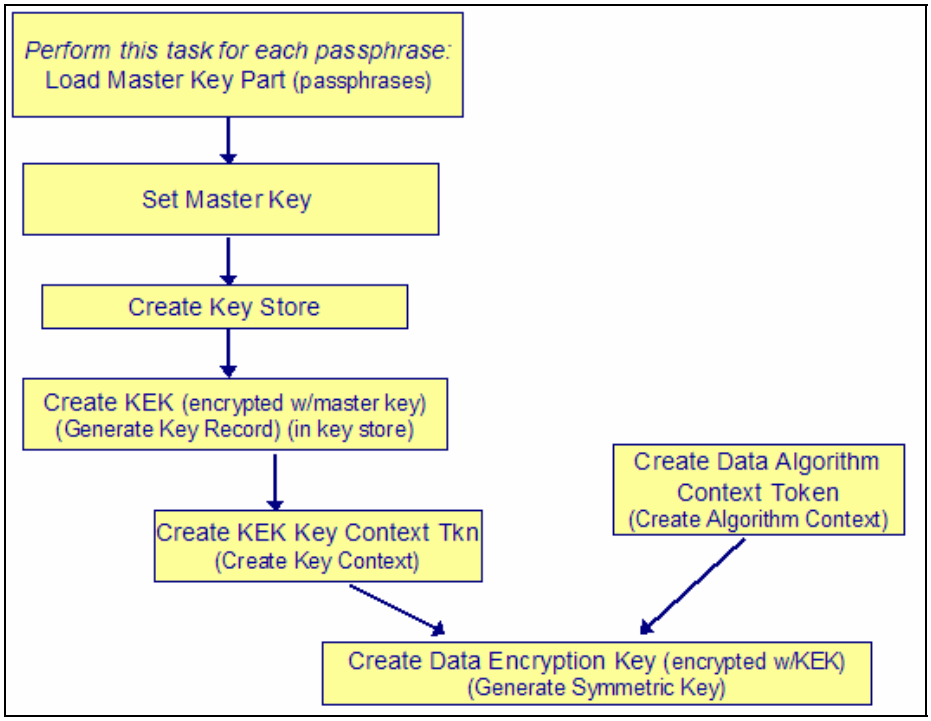

Figure 14. Generating a KEK and data-encryption key

Changing master-key values

Periodically, it is recommended to change master-key values. All keys that are encrypted under the master key must then be translated under the newly activated master-key value. If you neglect to do this, key values, as well as all data encrypted under them, might be lost.

You use the Translate Key Store API to translate key-store files from an old master-key value to an active master-key value. To translate a master-key-encrypted key that is located outside of key store, first use the Export Key API, and then use the Import Key API.

Master-key-encrypted keys that are not translated after the master key is changed might still be usable if you know the old master key's KVV. When a key is added to a key-store file, the KVV of the master key (under which the new key value is encrypted) is stored in the key record. When the key is used on a Cryptographic Services API, the API retrieves the KVV from the key record and compares it with the active and old master-key KVVs. However, for master-key-encrypted keys outside of key

store, you must manage the master-key KVV and supply it to any APIs yourself. By supplying a KVV, an API can detect when a key is encrypted under an old master-key value. The API runs as usual but returns a warning message specifying that you need to translate the key. If the key is not translated and the master-key value is changed again, the key value is lost, and you cannot recover it unless you restore the original value of the master key.

Master key variants

You can limit the use of a master-key-encrypted key by using master-key variants. For example, if your application program uses a key from key store to encrypt sensitive data, you can use variants to ensure that the key cannot be used to decrypt the data.

A master-key variant is a value that is “XOR’d” (that is, inserted through the use of an Exclusive Or [XOR] operation) into the master-key value before encrypting a key. You specify a master-key variant on an API (for example, the Generate Key Record API) by using the **Disallowed Function** parameter or field. Functions that you can disallow are Encrypt, Decrypt, MAC and HMAC, sign or any combination of these.

When a key that is encrypted under a master-key variant is used on an operation (for example, decrypting data), the variant must be supplied to decrypt the key properly. The master-key variant for a key-store file key is stored in the key record and picked up automatically when that key is specified on an API. For keys outside of key store, you must manage and supply the master-key variant yourself.

If the supplied variant indicates that the operation is disallowed, an error is returned. If the variant is altered to force the operation, the results will be bad. For example, if you use an altered variant on the Decrypt Data API, the decryption proceeds as usual, but the result is not clear-text data.

Migrating keys to another system

To move a master-key-encrypted key (in or outside of key store) to another system, use the Export Key API. The Export Key API translates the key from encryption under the master key to encryption under a KEK. On the target system, you can then use the Write Key Record API to move the migrated key into key store, or you can use the Import Key API to translate the key value to encryption under a master key. All these APIs require that the KEK information be specified through the use of an algorithm and key contexts.

The Export Key API is shipped with public authority *EXCLUDE. Be very careful about the access you give to the Export Key API. Anyone with access to master-key-encrypted keys and the Export Key API can obtain the clear-key values.

In general, you should not share master keys with another system. Each system should have unique master keys. However, to move an entire key-store file from one system to another without exposing clear-key values, you need to set up identical master-key values on both systems. To avoid exposing your master-key values, set up a temporary master key on both systems by loading and setting an unused master key with identical passphrases. On the source system, create a duplicate of the key-store file (for instance, by using the i5/OS CRTDUPOBJ CL command). Then, translate the duplicated key-store file to the temporary master key. After moving the key-store file to the target system, translate the key-store file to another master key.

Backing up keys

Keeping a current backup of all your keys is essential. In i5/OS V5R4, you must back up your master keys by saving their passphrases. Master-key passphrases should not be stored on the system in plain text. Also, do not encrypt them under any of the system's master keys or any key that is encrypted under a master key. If the master keys are lost (for example, when the LIC is installed) or damaged, you cannot recover the passphrases and, therefore, you cannot recover the master keys. Store the passphrases securely outside the system, such as in a safe. Additional help for saving and restoring master keys is planned for a future release of i5/OS.

Any time a new key is added to a key-store file, making a backup of the file is recommended. In addition, any time the key-store file is translated, you need to make a new backup of the file.

Do not forget about keys that are stored outside of key store. These should be backed up as well. Generally, these should not be encrypted under a master key. If you store a key outside of key store, it is best to encrypt it under a KEK. If you encrypt it under a master key and that master key is changed, you must remember to translate the key as described previously.

Key distribution

You can use the Cryptographic Services APIs in i5/OS V5R3 and V5R4 to implement a key negotiation protocol. Typically, you use symmetric-key algorithms to perform data encryption. The symmetric keys are distributed using asymmetric-key algorithms. The Cryptographic Services APIs support two asymmetric-key algorithms for use in key distribution:

- **RSA:** An RSA public key encrypts a symmetric key that is then distributed. The corresponding private key is used to decrypt it.
- **Diffie-Hellman (D-H):** The communicating parties generate and exchange D-H parameters, which are then used to generate key pairs. The public keys are exchanged, and each party is then able to compute the symmetric key independently.

A great deal of literature is available concerning key-exchange protocols, including setting up key negations and the many potential pitfalls. (See the references in Appendix B to learn more.)

SQL encryption and decryption

A previous example in this paper demonstrated the simplified interface that DB2 for i5/OS provides for encryption and decryption with SQL. Although the SQL functions provide a simple interface for encryption and decryption, there are no SQL functions to assist with the management of encryption keys.

This section contains more details on the syntax and other requirements for using the SQL built-in encryption and decryption functions.

Look at an example to understand how the DB2 encryption and decryption functions can be used to provide an extra layer of security:

```
SET ENCRYPTION PASSWORD = 'SEC01RET'
INSERT INTO customer VALUES('JOSHUA', ENCRYPT('1111222233334444'))

SET ENCRYPTION PASSWORD = 'SEC01RET'
SELECT name, DECRYPT_CHAR(card_nbr) FROM customer
```

In the preceding code, the Set Encryption Password statement provides DB2 with the key that is used to encrypt (and decrypt) the data. The next statement shows how the DB2 Encrypt function is used to encode the sensitive credit-card number before it is written into the DB2 table. The final statement shows the steps necessary to view the original credit-card number value of **1111222233334444**. First, the encryption key must be set to the same value that was used to encrypt the number. Then, one of the DB2 decryption functions must be used to convert the binary-encrypted value into the original character value.

Notice in this example that there are no SQL or DDS keywords telling DB2 to encrypt and decrypt the data automatically. Application changes are required because automatic encryption and decryption do not provide an extra layer of security. If DB2 automatically decrypts the credit-card number for all users that read the customer table, then the credit-card number is viewable to just as many users as it is without encryption. You can realize the security benefits of encryption only by changing your applications and interfaces to decrypt the data selectively for a subset of your authorized users. The benefit provided by the DB2 support is that these functions make it easy to encrypt and decrypt data. Your applications can invoke a simple SQL function instead of coding calls to complex cryptography APIs and services.

Preparing for encryption

Now that you understand how to use the SQL-encryption support, it is important to focus on the setup steps required by encryption. For several releases, OS/400 and i5/OS have had a set of cryptographic services (that is, algorithms) that have been available for applications and system functions such as SSL and IPSec to use. In i5/OS V5R3, access to these cryptographic services has been integrated into DB2 for i5/OS with the arrival of a set of SQL encryption and decryption scalar functions.

The DB2 encryption and decryption column functions use the encryption algorithms embedded in the no-charge IBM Cryptographic Access Provider 128-bit product (5722-AC3). As mentioned, this product no longer exists as of i5/OS V5R4 because cryptographic support is embedded into the base level of i5/OS. The IBM Cryptographic Coprocessor and Accelerator are not required and provide no benefit to the encryption algorithms used by the DB2 SQL functions. Instead of having to interface directly with OS/400 Cryptographic Services API, a developer can invoke a simple DB2 function as demonstrated below:

```
INSERT INTO employees VALUES( ENCRYPT_RC2('777-88-9999'), 'BOB SANDERS')
```

(**Note:** The DB2 encryption and decryption functions have been available since i5/OS V5R3. Starting with i5/OS V5R4, the ENCRYPT_TDES function for Triple DES encryption is also available.)

After ensuring that your i5/OS system has the needed prerequisites installed, the final preparation step is to identify the columns to encrypt. Data encryption requires changing the data type and length of your existing column definitions. The data type must be changed because an encrypted value is a binary string that can only be stored in a column that is defined with one of the following SQL data types:

- BINARY
- VARBINARY
- CHAR FOR BIT DATA
- VARCHAR FOR BIT DATA
- BLOB

If the target DB2 table is defined with DDS, then the target column must be defined as a fixed or varying-length character field with a coded character set identifier (CCSID) value of 65535.

The length of an existing column must increase because some overhead bytes are needed for the encrypted value. The DB2 engine uses these extra bytes to store information on the attributes (for example, CCSID) of the data that is encrypted in the column and the algorithm that encrypted the data. The data stored in these extra bytes allows DB2 for i5/OS to share and exchange encrypted data with other DB2 server products. This length must increase if a password hint is stored with the encrypted value. (Password hints are covered later.) The length of an encrypted string value is the actual length of the string plus *N* bytes. *N* is defined as follows:

- If a hint is not specified, *N* is 8 bytes (or 16 bytes if the input string is defined as large object (LOB), BINARY or VARBINARY or the input string and password have different CCSID values) plus the number of bytes to the next 8-byte boundary.
- If a hint is specified, *N* is 8 bytes (16 bytes if the input string is defined as LOB, BINARY or VARBINARY or the input string and password have different CCSID values) plus the number of bytes to the next 8-byte boundary plus 32 bytes for the hint length.

Different CCSID values can result because the password string is assigned the CCSID that is used by the job, while the encrypted value is assigned the CCSID of the result data type. The iSeries SQL Reference provides more details on length requirements (http://www.ibm.com/eserver/iseries/db2/books.htm).

Here are a couple of examples to better understand the length requirements:

- A 16-digit credit-card number that is stored in a VARBINARY column with no hint requires a column length of 32 bytes. That is 16 bytes for nonencrypted data plus 16 extra bytes plus 0 extra bytes (already at an 8-byte boundary).
- A 9-digit Social Security number that is stored in a VARCHAR FOR BIT DATA column with a hint requires a column length of 56 bytes. That is 9 bytes for nonencrypted data plus 8 extra bytes plus 7 extra bytes (to reach the next 8-byte boundary) plus 32 bytes for the hint.

Besides minimizing the number of column-definition changes, performance overhead is another reason to encrypt only a small subset of columns. For instance, if you increase your office building security by adding a badge reader to the front door, it takes a little longer to get to your desk each morning. Slower database access is a result of adding encryption and decryption to your application. Table 1 shows the results of a simple comparison test done in the lab with a single-column table to compare the speed of inserting and retrieving a single row.

	Without encryption	With encryption
SELECT statement	0.1 millisecond	3.0 milliseconds
INSERT statement	0.05 millisecond	1.1 milliseconds

Table 1. Comparing the speed of inserting and retrieving a single row

The encryption recipe

Now that the preparations have all been made, it is time to learn how to use the DB2 Encrypt function. The first step for data encryption is providing DB2 with an encryption key. The encryption password can be set at a job or connection level with the Set Encryption Password statement. The statement is a good method to use when the same password must be shared across rows in a table or across different tables.

The encryption password is a character string that must have a length that is between 6 and 127 bytes, and the value is case-sensitive. To help administrators and programmers remember the password value,

the Set Encryption Password statement lets you store an optional hint (up to 32 bytes) with the encrypted data value. The following example shows how to supply DB2 with the encryption key and a hint:

```
CREATE TABLE emp (
  SocialSecurity VARCHAR(56) FOR BIT DATA,
  name VARCHAR(30))

SET ENCRYPTION PASSWORD= 'at95lantic' WITH HINT='ocean'

INSERT INTO emp VALUES(ENCRYPT_RC2('123456789'), 'JENNA')

SELECT GETHINT(socialSecurity) FROM emp
```

The Set Encryption Password statement specifies an encryption key value of 'at95lantic' with a hint value of 'ocean'. The numeric digits, 95, are in the middle of password string because just having a word such as ‘atlantic’ as a password is not a good idea. A hacker could programmatically try all the words in the dictionary in a short period of time.

When the Insert statement encrypts the employee’s Social Security number, DB2 also stores the hint value along with the encrypted value in that column. If an administrator ever forgets the encryption key, he can use the DB2 GetHint function to retrieve the hint value of 'ocean' to remind himself that the name of an ocean was used for the encryption key. The encryption key specified on the Set Encryption Password statement is active until the job or activation group ends.

You can also vary the encryption password across different rows in a table by running the Set Password statement multiple times or by specifying the password-string value on the DB2 Encrypt functions. The Encrypt function supports three input parameters, but the only required parameter is the input data string. The optional parameters are the password and hint. Here is an example of how to perform the same encryption as in the preceding example:

```
INSERT INTO emp
  VALUES(ENCRYPT_RC2('123456789','at95lantic', 'ocean'),'JENNA')
```

The capability to specify different passwords for each row might be useful for a Web application where all customers are given the ability to encrypt their own credit-card number. When customers input their credit-card number, they also specify a password that is used on the ENCRYPT_RC2 function. That password is used to encrypt their card number as it is written to the back-end database. These SQL encryption examples look exactly the same if the ENCRYPT_RC2 function is replaced with the ENCRYPT_TDES function.

Encryption password management and distribution

Obviously, it is not a good idea to hard code the encryption password and hint in your application source code. The recommended approach is to supply these values through host variables or parameter markers so that the encryption password is less visible to anyone who scans the application source code:

```
SET ENCRYPTION PASSWORD=:hostvar1 WITH HINT=:hostvar2
INSERT INTO emp VALUES(ENCRYPT('123456789',?,?), 'JENNA')
```

If the application manages the encryption and decryption of the data, you can make the password value more secure by storing it in a validation-list object or by using the i5/OS V5R4 APIs for key management.

Decryption: unlocking the data

The final step in this data-protection process is getting the original data value back with the DB2 decryption function. i5/OS V5R3 includes the following set of decryption functions:

- DECRYPT_CHAR
- DECRYPT_DB
- DECRYPT_BIT
- DECRYPT_BINARY

Because they return a character string to the invoker, the DECRYPT_CHAR and DECRYPT_DB functions are probably the most commonly used. As with the Encrypt function, the Decrypt functions accept up to three parameters, but only the first one is required. The first parameter obviously identifies the encrypted value that needs to be decrypted. The second parameter again can be used to specify the password, and the final parameter is an integer value that you can use to specify the CCSID value for any character-string values that are returned.

For the decryption functions to return the original value, the encryption password must be set to the same password value that was used to encrypt the data with the Encrypt function. If the decryption password is different, a runtime error is returned. Here is an example showing how to decrypt the data that was encrypted in the earlier example:

```
SET ENCRYPTION PASSWORD='at66lantic'
SELECT DECRYPT_CHAR(socialSecurity) FROM emp WHERE name='JENNA'
```

Any application that reads the Social Security number from the employee table without the decryption function receives the encrypted binary-string value by default.

As you can see, invoking the DB2 decrypt functions is simple.

Native interfaces and DB2 encryption

So far in this section of the paper, all of the examples have been written in SQL; therefore, you might be wondering if applications that use the native (non-SQL) interface can benefit from this DB2 support. The answer is yes. You do need to use some SQL in the process, but that does not mean rewriting your applications to use SQL. Native users must understand that the underlying table that contains the encrypted data does not have to be created with SQL. The column being encrypted just needs to satisfy the previously discussed requirements.

Database triggers provide an excellent way of making encryption services available without changing all of your applications to use SQL. Triggers are useful whether or not you use the DB2 encryption and decryption functions to call the Cryptographic Services APIs directly.

You can define the Before Insert and Update triggers to intercept the Write requests to your database and then Encrypt any sensitive column data before it is passed to the DB2 engine. Using this trigger approach means that the only program that must use SQL to invoke one of the SQL built-in encryption functions is the trigger program. You can use both SQL and external triggers.

The following example uses SQL triggers to encrypt Social Security numbers that are inserted into the employee table:

```
CREATE TRIGGER protect_ssnumber
  BEFORE INSERT ON emp
REFERENCING NEW ROW AS n
FOR EACH ROW
BEGIN
  DECLARE encrypt_key VARCHAR(127);
  SET encrypt_key = get_passwordUDF('EMP');
  SET n.socialSecurity = ENCRYPT_TDES(n.socialSecurity, encrypt_key);
END
```

Any native Write (or SQL Insert) into the employee table causes this BEFORE INSERT trigger to be invoked. When the trigger is invoked, the protect_id SQL trigger first calls the user-defined get_passwordUDF() function to get the encryption password for the employee table. That password is then used to encrypt the Social Security number in the Before record image that DB2 uses to generate a new row into the employee table.

SQL views are probably the most common approach to give native programs the ability to read the decrypted version of sensitive data. Here is an example view that can be used in this manner:

```
CREATE VIEW my_logins(system, login, passwd) AS
  SELECT system,login,char(decrypt_char(passwd),6 )
  FROM regusers WHERE userid=USER
```

The SQL my_logins view can then be opened as a nonkeyed logical file by the native program, and the decrypt column function contained in the SQL view allows the unencrypted version of the column data to be returned to the native program. The decryption occurs only if the native program has set the proper encryption key either directly (by running the Set Encryption Password SQL statement) or indirectly (by calling another program to perform the password activation). Encrypted columns require a longer column length; therefore, native programs need to decrypt the data or be modified to handle the longer column length when the encrypted data is returned.

Using an SQL View to decrypt data automatically simplifies the decryption process, but this same SQL View also provides a mechanism for allowing system users to retrieve sensitive data in a bulk fashion. Data-security concerns must be reviewed carefully before using this view-based approach for decryption.

Integrated encryption and instead Of triggers

DB2 includes support in i5/OS V5R3 and V5R4 for a new type of trigger that makes it even easier to incorporate encryption and decryption into your applications. This new type of trigger is known as an Instead Of trigger. (**Note:** The i5/OS V5R3 Instead Of Trigger PTF support does include some restrictions [see http://www.ibm.com/iseries/db2/iot.html for more details] that were eliminated in i5/OS V5R4. In addition, the most recent i5/OS V5R3 Database Group PTF is required for Instead Of trigger support.

The biggest difference with Instead Of triggers (compared to the existing DB2 trigger support) is that Instead Of triggers can only be defined over SQL views. IBM created Instead Of triggers to enhance the behavior of Insert, Update and Delete operations against views. Certain views contain transformations that make the view read-only. For example, Insert and Update operations cannot be performed against the my_logins view because the decrypt_char scalar function causes this view to be treated as read-only.

The my_logins view is used by applications so that the password column can be automatically decrypted on Read operations. It makes sense that, if an application performs Write operations against this view,

the my_logins view will automatically encrypt the data. You can define an Instead Of trigger on the my_logins view to perform this automatic data encryption. Here is an example of how to use Instead Of triggers to perform the encryption when Insert and Update operations reference the my_logins view:

```
CREATE TRIGGER insert_my_logins
  INSTEAD OF INSERT ON my_logins
  REFERENCING NEW AS n
    FOR EACH ROW MODE DB2SQL
 INSERT INTO regusers
         VALUES(USER,n.system,n.login,Encrypt(n.passwd))

CREATE TRIGGER update_my_logins
  INSTEAD OF UPDATE ON my_logins
  REFERENCING OLD AS o NEW AS n
    FOR EACH ROW MODE DB2SQL
  UPDATE reguser SET system=n.system,
    login=n.login, passwd=Encrypt(n.passwd)
  WHERE system=o.system AND login=o.login AND userid=USER
```

When an Insert is performed against the my_logins view, DB2 calls the insert_my_logins trigger instead of signaling a read-only view error. The insert_my_logins trigger grabs the password string (n.passwd) from the Insert against the view and then runs an Insert statement against the underlying table (regusers) so that the password string is encrypted. The update_my_logins trigger operates in a similar fashion.

You cannot use the following clauses on the Create Trigger statement when creating an Instead Of trigger:

- Before and After clauses
- Of column-name clause for an Update trigger
- For Each Statement clause
- When clause

For any given view, only one Instead Of trigger can be defined for each operation (Insert, Update and Delete) meaning a view can have a maximum of three Instead Of triggers. Instead Of triggers cannot be defined over a system or catalog view (for example, SYSTABLES in QSYS2).

Query optimization and performance considerations

In some cases, creating indexes on encrypted data is a good idea. Exact matches and joins of encrypted data use the indexes you create. Because encrypted data is essentially binary data, range checking of encrypted data requires table scans and is to be avoided as much as possible.

In cases where searches must be performed against an encrypted column, you need to use the Encrypt functions in a way that minimizes the number of values that are decrypted in the search. For example, the optimal way to search for a user by her Social Security number is to use the following query, because it avoids decrypting every Social Security number in the table:

```
SELECT name FROM emp WHERE socialSecurity = ENCRYPT_TDES('111222333');
```

Because extra processing takes place for encryption and decryption, these data-security operations slow most SQL statements. In general, encryption should only be used for a small set of sensitive columns, such as patient name and Social Security number. (**Note:** If the WHERE clause (in the example above) is changed to DECRYPT_CHAR(socialSecurity)='111222333', then every row in the employee table must be decrypted. For queries joining on two encrypted columns, just use the encrypted values to perform the join; there is no need to decrypt the column values first.)

Encryption implementation considerations

This reference section suggests considerations and offers recommendations that regard your implementation of encryption for DB2 for i5/OS data.

Column encryption considerations

If encrypting data that is currently stored in a column (or field) within a DB2 table (or physical file), then changing the column definition is one of the first steps. An encrypted value is a binary string; therefore, the column data type must be changed to a data type that supports binary strings. If the encrypted binary string is stored in a normal character column, then an application runs the risk of DB2 converting the string to a code page or character set. The following SQL data types can be used to store binary values:

- BINARY
- VARBINARY
- CHAR FOR BIT DATA
- VARCHAR FOR BIT DATA
- BLOB

If the target DB2 table is defined with DDS, then the target column must be defined as a fixed or varying-length character field with a CCSID value of 65535.

Depending on the encryption algorithm used to encrypt the data, the length of an existing column might not have to be increased. As noted earlier, usage of the SQL ENCRYPT_RC2 or ENCRYPT_TDES function requires that the column length be increased.

Now that the possible impacts to the column definitions in your database are clear, how do you determine which columns to encrypt? The simple answer is that encryption should be reserved only for those columns that contain personal or sensitive data. A credit-card number is an example of sensitive data that should be encrypted to protect that value. In contrast, a column containing a customer's name is probably not an encryption candidate because that name is widely available from other sources, even though a last name is considered personal data. Encrypting the data in all columns is not recommended.

Performance and sorting restrictions are the biggest reasons for limiting encryption to a small number of columns. First, additional system processing is consumed and required when encrypting and decrypting data. Second, performance problems can also surface when the application wants to search on a specific value. As documented in the "SQL encryption" section of this paper, the application should give the query the encrypted version of the search value because this allows DB2 to search for the value without first decrypting all of the rows in the table. Finally, the data stored in an encrypted column cannot be sorted directly in a meaningful way. If the U.S. Social Security Number (SSN) data in a column is stored in an encrypted fashion, then any program that displays customer data sorted by SSN needs to be changed because the encrypted versions of the SSN will sort differently. The program has to be changed to decrypt all of the values into some temporary structure that can then make the decrypted values available for the sorting request. This obviously slows the performance of this request. Therefore, careful consideration must be given before encrypting a column that is frequently referenced in sorting or lookup operations by an application.

After the columns have been identified for encryption, one final option to consider is moving the encrypted columns to a separate table. Although this option can require extensive application changes, it does help

limit access to the sensitive data stored in the encrypted column. For example, if the column containing credit-card data is encrypted and moved to a separate table, any existing applications that access the customer table no longer have direct access to the credit-card column. Therefore, any retrieval of the credit-card data requires the application to be changed explicitly to call a program or access the new table containing the credit-card data. In addition, applications that only access nonsensitive customer data do not have to be enhanced with the ability to decrypt data.

Cryptographic services tips

What is the best cryptographic algorithm? What is a good key size? Cryptographic application programmers often ask these kinds of questions. The answers really depend on application requirements. But generally, the following advice serves well:

- Use the Advanced Encryption Standard (AES). AES is the new official standard and runs faster than the Data Encryption Standard (DES) and Triple DES (3DES). Do not use DES, if possible.
- Use a 256-bit key. Because of collision attacks, a key of *n* bits actually gives you n/2-bit security. In other words, it takes a workload of about 2^{128} steps to break a 256-bit key.
- Use cipher-block-chaining (CBC) mode to mask patterns in the data.
- A 256-bit (32-byte) block size is best because a large block size reduces the number of ciphertext collisions, which leak information. However, the AES standard supports only a 128-bit (16-byte) block size. Therefore, when using a 256-bit key with a 128-bit block size, limiting how much total data is encrypted with a single key is important. (Otherwise, you might as well use a smaller key size.) With CBC mode, you should limit it to 2^{32} blocks or so.
- To avoid identical ciphertext in the first block, you must use a substantially different Initialization Vector (IV) for each message. One option is to use the Generate Pseudo-random Number API to generate a unique IV value. Alternatively, you can use a unique number (known as a nonce), but you must encrypt it before using it as an IV value, and you must ensure that, for a given key, the counter never wraps. The nonce method has advantages for communications in that you can include the information needed to reconstruct the nonce with your ciphertext message in typically 4 to 8 bytes (compared to 32 bytes, if using a randomly-generated IV value). The recipient can then reject any message numbers that are not greater than the last message received. (This applies only to encryption. For MAC operations, a null IV value is fine.)
- You need to use encryption with a separate authentication function. Even if modified messages decrypt to nonsense, they can still cause damage. (Why? If an encrypted message is modified, the decryption operation does not always fail. The decryption operation completes as usual, but the results are garbage. An application program cannot always detect this. For example, an application might use bytes 7 and 8 of the decrypted message as a numeric value. How does the application program know that the value of 43 382 should be 200?) Do the authentication first, then the encryption. Do not use the same key for encryption and authentication.
- Simple attacks can be performed on the MAC of a plain-text message. To prevent these attacks, concatenate the length of the input data to the start of the message, then perform a MAC operation on the entire string.
- Use a strong Secure Hash Algorithm, such as SHA-256 or SHA-512, if possible. (Do not bother using SHA-384. The underlying function performs all the work of SHA-512 and then throws away part of the results.) The i5/OS V5R4 Calculate HMAC API now supports these algorithms.

- Because of weaknesses in the SHA-hash functions, you must always perform a double hash (that is, hash the data, then hash the result). This is unnecessary for HMAC.
- Use SHA-256 HMAC for the authentication function, and use all 256 bits (32 bytes) as the MAC value, if possible.

Remote clients and encryption keys

If a remote client (that is, a Java applet running in a Web browser) sends an encryption key to encrypt or decrypt data on a System i model, then extra precautions must be taken when passing this key to the system. To protect the encryption password in these cases, consider using a communications-encryption mechanism, such as IPsec (or SSL if connecting between System i models).

This protection is needed when the remote client specifies the encryption password on the invocation of the SQL encryption and decryption functions. In these cases, the encryption password key is sent “in the clear” to DB2 for i5/OS. The password itself is not encrypted.

Summary

This paper has demonstrated how the new encryption and decryption functions in IBM DB2 for i5/OS provide a simple way to add an extra lock of security around sensitive data. Before implementing column encryption, careful consideration needs to be given to how your application and business processes are going to be changed to address selective decryption of the encrypted columns in your database. Although this task is not trivial, the requirement for encrypting personal and financial data is one that cannot be ignored.

Appendix B: Resources

These Web sites provide useful references to supplement the information contained in this document:

- IBM eServer iSeries Information Center
  http://ibm.com/iseries/infocenter
  - Cryptographic Coprocessors for iSeries
    http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/rzahu/rzahucryptocardconcept.htm
  - 2058 Cryptographic Accelerator for iSeries
    http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/rzajc/rzajcaccel2058.htm
  - OS/400 and 2058 Cryptographic Function Comparison
    http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/apis/qc3Compare.htm
  - Cryptographic Services APIs
    http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/apis/catcrypt.htm
  - Cryptographic Services Master Keys
    http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/apis/qc3MasterKeys.htm
  - Cryptographic Services Key Store
    http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/apis/qc3KeyStore.htm
  - Digital Certificate Manager
    http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/rzahu/rzahurazhudigitalcertmngmnt.htm
- IBM System p and AIX Information Center
  http://publib.boulder.ibm.com/infocenter/pseries/index.jsp
- IBM Publications Center
  http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi?CTY=US
- IBM Redbooks™
  http://www.redbooks.ibm.com/

IEncryption programming examples with RPG

- Scenario: Key Management and File Encryption Using the Cryptographic Services APIs (i5/OS V5R3)
  http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/info/apis/qc3Scenario.htm
- Scenario: Key Management and File Encryption Using the Cryptographic Services APIs (i5/OS V5R4)
  http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/apis/qc3Scenario.htm
- SystemiNetwork.com
  - APIs by Example: Cryptographic Services APIs (multipart series)
  - Triple-DES Encryption from RPG

Further reading

- *Practical Cryptography*, Bruce Schneier (0-471-11709-9)
- *Applied Cryptography*, Bruce Schneier (0-471-11709-9)
- *Cryptography Decrypted*, H. X. Mel and Doris Baker (0-201-61647-5)
- PKCS #5: Password-Based Cryptography Standard
  http://rsasecurity.com/rsalabs/node.asp?id=2127
- Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management
  http://ietf.org/rfc/rfc1422.txt?number=1422

Appendix C: About the author

Beth Hagemeister is an IBM System i software engineer in Rochester with 20 years of experience developing cryptographic software for System/38, AS/400 and System i systems.

Kent Milligan is a DB2 technology specialist in IBM ISV Business Strategy and Solutions Enablement for the IBM System i platform. After graduating from the University of Iowa, Kent spent the first eight years of his IBM career as a member of the DB2 development group in Rochester, Minnesota. He speaks and writes regularly on various DB2 for i5/OS relational database topics.

Appendix D: Acknowledgements

Much of this content comes from articles originally published in 2005 and 2006 by SystemiNetwork.com. Thanks to Thomas Barlen of IBM for his contributions.

Trademarks and special notices

AIX, AS/400, DB2, eServer, i5/OS, IBM, the IBM logo, iSeries, OS/400, Redbooks, System/38, System I, System p, System x and System z are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.

Some information addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.