Database Encryption in Oracle9*i*™

An Oracle Technical White Paper

February 2001

Database Encryption in Oracle9*i*

SECURING SENSITIVE INFORMATION

The Internet poses new challenges in information security, especially for those organizations seeking to become e-businesses. Many of these security challenges can be addressed by the traditional arsenal of security mechanisms:

- strong user authentication to identify users
- granular access control to limit what users can see and do
- auditing for accountability
- network encryption to protect the confidentiality of sensitive data in transmission

Encryption is an important component of several of the above solutions. For example, Secure Sockets Layer (SSL), an Internet-standard network encryption and authentication protocol, uses encryption to strongly authenticate users, via X.509 digital certificates. SSL also uses encryption to ensure data confidentiality, and cryptographic checksums to ensure data integrity. Many of these uses of encryption are relatively transparent to a user or application. For example, many browsers support SSL, and users generally don't need to do anything special to enable SSL encryption.

Oracle has provided network encryption between database clients and the Oracle database since Oracle7. Oracle Advanced Security, an option to Oracle9*i*, provides encryption and cryptographic integrity check for any protocol supported by Oracle9*i*, including Net8, Java Database Connectivity (JDBC) (both "thick" and "thin" JDBC), and the Internet Intra-Orb Protocol (IIOP). Oracle Advanced Security also supports SSL for Net8, "thick" JDBC and IIOP connections.

While encryption is not a security cure-all, it is an important tool used to address specific security threats, and will become increasingly more important with the growth of e-business, most particularly in the area of encryption of stored data. For example, while credit card numbers are typically protected in transit to a web site using SSL, the credit card number is often stored in the clear (unencrypted), either on the file system, where it is vulnerable to whomever can break into the host and gain root access, or in databases. While databases can be made quite secure through proper configuration, they can also be vulnerable to host break-ins if the host is misconfigured. There have been several recent, well-publicized break-ins, in which a hacker obtained a large list of credit card numbers by breaking into a database.

Encryption of stored data thus represents a new challenge for e-businesses, and can be an important tool in dealing with specific types of security threats.

ENCRYPTION ISSUES

While there are many good reasons to encrypt data, there are many bad reasons to encrypt data. Encryption does not solve all security problems, and may even make some problems worse. The following section describes some of the misconceptions about encryption of stored data.

Issue 1: Encryption is not access control

Most organizations need to limit access to data to those who have a "need to know." For example, a human resources system may limit employees to reviewing only their own employment records, while managers of employees may see the employment records of those employees working for them. Human resources specialists may also need to see employee records for multiple employees.

This type of security policy — limiting data access to those with a need to see it — is typically addressed by access control mechanisms. The Oracle database has provided strong, independently-evaluated access control mechanisms for many years. Recently, Oracle8*i* has added the ability to enforce access control to an extremely fine level of granularity, through its Virtual Private Database capability.

Because human resources records are considered sensitive information, it's tempting to think that this information should all be encrypted "for better security." However, encryption cannot enforce the type of granular access control described above, and may actually hinder data access. In the human resources example, an employee, his manager, and the HR clerk all need to access the employee's record. If employee data is encrypted, then each person also has to be able to access the data in unencrypted form. Therefore, the employee, the manager and the HR clerk would have to share the same encryption key to decrypt the data. Encryption would therefore not provide any additional security in the sense of better access control, and the encryption might actually hinder the proper functioning of the application. There is the additional issue that it is very difficult to securely transmit and share encryption keys among multiple users of a system.

A basic principle behind encrypting stored data is that it must not interfere with access control. For example, a user who has SELECT privilege on EMP should not be limited by the encryption mechanism from seeing all the data he is otherwise allowed to see. Similarly, there is little benefit to encrypting, (for example) part of a table with one key and part of a table with another key if users need to see all encrypted data in the table; it merely adds to the overhead of decrypting data before users can read it. Provided that access controls are implemented well, there is little additional security provided within the database itself from encryption; any user who has privilege to access data within the database has no more nor less privilege as a result of encryption. *Therefore, encryption should never be used to solve access control problems.*

Issue 2: DBAs can access all data

Some organizations are concerned that database administrators (DBAs), because they typically have all privileges, are able to see all data in the database. These organizations feel that the DBAs should merely administer the database, but should not be able to see the data that the database contains. Some organizations are also concerned about the concentration of privilege in one person, and would prefer to partition the DBA function, or enforce two-person rules.

It's tempting to think that encrypting all data (or significant amounts of data) will solve the above problems, but there are better ways to accomplish these objectives. First of all, Oracle does support limited partitioning of DBA privilege. Oracle9*i* provides native support for SYSDBA and SYSOPER

users. SYSDBA has all privileges, but SYSOPER has a limited privilege set (e.g. startup and shutdown of the database). Furthermore, an organization can create smaller roles encompassing a number of system privileges. A JR_DBA role might not include all system privileges, but only those appropriate to a more junior database administrator (such as CREATE TABLE, CREATE USER, etc.) Oracle does not audit the actions taken by SYS (or SYS-privileged users) but does audit startup and shutdown of the database in the operating system records.

Furthermore, the DBA function by its nature is a trusted position. Even organizations with the most sensitive data — such as intelligence agencies — do not typically partition the DBA function. Instead, they vet their DBAs strongly, because it *is* a position of trust.

Encryption of stored data must not interfere with the administration of the database, or larger security issues can result than you were attempting to address with encryption. For example, if by encrypting data, you corrupt the data, you've created a security problem: data is not meaningful and may not be recoverable.

Encryption can be used to mitigate the ability of a DBA — or other privileged user — to see data in the database, but it is not a substitute for vetting a DBA properly, or for limiting the use of powerful system privileges. If an untrustworthy user has significant privilege, there are multiple threats he can pose to an organization, which may be far more significant than viewing unencrypted credit card numbers.

Issue 3: Encrypting everything does not make data secure

It's a pervasive tendency to think that if storing some data encrypted strengthens security, then encrypting everything makes all data secure.

We've already seen why encryption does not address access control issues well. Consider the implications of encrypting an entire production database. All data must be decrypted to be read, updated, or deleted, and, as discussed earlier, the encryption must not interfere with normal access controls. Encryption is innately a performance-intensive operation; encrypting all data will significantly affect performance. Availability is a key aspect of security and if, by encrypting data, you make data unavailable, or the performance adversely affects availability, you have created a new security problem.

Encryption keys must be changed regularly as part of good security practice, which necessitates that the database be inaccessible while the data is being decrypted and reencrypted with a new key or keys. This also adversely affects availability.

While encrypting all or most data in a production database is clearly a problem, there may be advantages to encrypting data stored off-line. For example, an organization may store backups for a period of six months to a year off-line, in a remote location. Of course, the first line of protection is to secure the data in a facility to which access is controlled, a physical measure. However, there may be a benefit to encrypting this data before it is stored, and since it is not being accessed on-line, performance need not be a consideration. While Oracle9*i* does not provide this facility, there are vendors who can provide such encryption services. Organizations considering this should thoroughly test that data (that is encrypted before off-line storage) can be decrypted and re-imported successfully before embarking on large-scale encryption of backup data.

SOLUTIONS FOR STORED DATA ENCRYPTION IN ORACLE

Oracle9*i* Data Encryption Capabilities

While there are many security threats that encryption cannot address well, it is clear that one can obtain an additional measure of security by selectively encrypting sensitive data before storage in the database. Examples of such data could include:

- credit card numbers
- national identity numbers
- passwords for applications whose users are not database users

To address the above needs, Oracle8*i* (release 8.1.6) introduced a PL/SQL package to encrypt and decrypt stored data. The package, DBMS_OBFUSCATION_TOOLKIT, is provided in both Standard Edition and Enterprise Edition Oracle9*i*. The package is documented in the Oracle9*i* Supplied PL/SQL Packages Reference Guide.

The package currently supports bulk data encryption using the Data Encryption Standard (DES) algorithm, and includes procedures to encrypt (DESEncrypt) and decrypt (DESDecrypt) using DES. The package does not currently support the Advanced Encryption Standard, the successor algorithm to DES, though this is planned for a future release of Oracle9*i*.

Key management is programmatic, that is, the application (or caller of the function) has to supply the encryption key, which means that the application developer has to find a way of storing and retrieving keys securely. The relative strengths and weaknesses of various key management techniques are discussed later in this paper. The DBMS_OBFUSCATION_TOOLKIT package, which can handle both string and raw data, requires the submission of a 64-bit key. The DES algorithm itself has an effective key length of 56-bits. The DBMS_OBFUSCATION_TOOLKIT package is granted to PUBLIC by default.

Oracle has added support for triple DES (3DES) encryption in Oracle8*i* release 8.1.7. The DBMS_OBFUSCATION_TOOLKIT package includes additional functions to encrypt and decrypt using 2-key and 3-key 3DES, in outer cipher-block-chaining mode. They will require key lengths of 128 and 192 bits, respectively.

Oracle8*i* release 8.17 also added support for cryptographic checksumming using the MD5 algorithm (using the MD5 procedure of the DBMS_OBFUSCATION_TOOLKIT package). Cryptographic checksums can ensure data integrity; that is, that data has not been tampered with. For example, an organization concerned that users not change salary values randomly could store a checksum of salary values in a separate table. Only users changing the salary through an application (e.g. through executing a procedure) would also have the privileges to insert a checksum for the new salary into a salary audit table.

Partner Applications

Organizations seeking a more robust implementation of encrypting stored data in the Oracle database can consider a product offering from Oracle partners such as Protegrity. Protegrity has provided database encryption capabilities since Oracle8. In the Protegrity solution, key management is automatic. There are multiple customers using it in production who are happy with the functionality.

Performance may be an issue in the Protegrity product, depending on how much data the customer wants to encrypt. Since Protegrity has built their solution using the extensibility features of the server, their "data cartridge" does not run in the server address space, and thus does not perform as well as a native implementation of encryption in Oracle would. The tradeoff of more automatic key management may well be worth it for Oracle customers, however.

CHALLENGES OF ENCRYPTION

This paper has already discussed why encryption should not be used to address threats better addressed by access control mechanisms, and some of the reasons why encryption is not a security cure-all. Even in cases where encryption can provide additional security, it is not without technical challenges, as described in the following sections.

Encrypting Indexed Data

Special difficulties arise in handling encrypted data which is indexed. For example, suppose a company uses national identity number (e.g. U.S. Social Security Number (SSN)) as the employee number for its employees. The company considers employee numbers to be very sensitive data, and the company therefore wants to encrypt data in the EMPLOYEE_NUMBER column of the EMPLOYEES table. Since EMPLOYEE_NUMBER contains unique values, the database designers want to have an index on it for better performance.

If the DBMS_OBFUSCATION_TOOLKIT (or another mechanism) is used to encrypt data in a column, then an index on that column will also contain encrypted values. While the index can still be used for equality checking (i.e. 'SELECT * FROM emp WHERE employee_number = '123245'), the index is essentially unusable for other purposes. Oracle therefore recommends that developers not encrypt indexed data (and in fact, we do not support encrypting indexed data).

In the above example, a company that wants to encrypt social security number (or national identity number) could create an alternate unique identifier for its employees, create an index on this employee number, but retain the employee number in clear text. The national identity number could be a separate column, and the values encrypted therein by an application, which would also handle decryption appropriately. The national identity number could be obtained when necessary, but would not be used as a unique number to identify employees.

Given the privacy issues associated with overuse of national identity numbers (for example, identity theft), the fact that some allegedly unique national identity numbers have duplicates (US Social Security Numbers), and the ease with which a sequence can generate a unique number, there are many good reasons to avoid using national identity numbers as unique IDs.

Binary Large Objects (BLOBS)

Certain datatypes require more work to encrypt. For example, Oracle supports storage of binary large objects (BLOBs), which lets users store very large objects (e.g. gigabytes) in the database. A BLOB can be either stored internally as a column, or stored in an external file.

To use the DBMS_OBFUSCATION_TOOLKIT, the user would have to split the data into 32767 character chunks (the maximum that PL/SQL allows) and then would have to encrypt the chunk and append it to the BLOB. To decrypt, the same procedure would have to be followed in reverse.

Key Management

Key management, including both generation of and secure storage of cryptographic keys, is arguably one of the most important aspects of encryption. If keys are poorly-chosen or stored improperly, then it makes it far easier for an attacker to break the encryption. Rather than using a "brute force" attack (that is, cycling through all the possible keys in hopes of finding the correct decryption key), cryptanalysts often seek weaknesses in the choice of keys, or the way in which keys are stored.

Key generation is an important aspect of encryption. Typically, keys are generated automatically through a random-number generator, from a cryptographic seed. Provided that the random number generation is sufficiently strong, this can be a secure form of key generation. However, if random numbers are not well-formed, but have elements of predictability, the security of the encryption may be easily compromised. Netscape had a well-publicized vulnerability in their SSL implementation several years ago when it was discovered that two of the three elements of their random number generation were not random (e.g. machine serial number and time of day). The encryption key for SSL sessions had an effective key length of 9 bits, rather than the advertised 40 bits, because of the weakness of the key generation. An SSL session key could be easily broken, not because the encryption algorithm was weak, but because the key was easily derived.

To address the issue of secure cryptographic key generation, Oracle9*i* adds support for a secure random number generation, the GetKey procedure of the DBMS_OBFUSCATION_TOOLKIT. The GetKey procedure calls the secure random number generator (RNG) that has previously been certified against the Federal Information Processing Standard (FIPS)-140 as part of the Oracle Advanced Security FIPS-140 evaluation.

Developers should *not* use the DBMS_RANDOM package. The DBMS_RANDOM package generates pseudo-random numbers; as RFC-1750 states, "The use of pseudo-random processes to generate secret quantities can result in pseudo-security."

Key Transmission

If the key is to be passed by the application to the database, then it must be encrypted. Otherwise, a snooper could grab the key as it is being transmitted. Use of network encryption, such as that provided by Oracle Advanced Security, will protect all data in transit from modification or interception, including cryptographic keys.

Key Storage

Key storage is one of the most important, yet difficult, aspects of encryption. To recover data encrypted with a symmetric key, the key must be accessible to the application or user seeking to decrypt data. The key needs to be easy enough to retrieve that users can access encrypted data, without significant performance degradation. The key needs to be secure enough that it's not easily recoverable by someone trying to maliciously access encrypted data he is not supposed to see.

The three basic options available to a developer are:

- store the key in the database
- store the key in the operating system
- have the user manage the key

Storing the keys in the database cannot always provide "bullet-proof" security if you are trying to protect data against the DBA accessing encrypted data (since an all-privileged DBA could access tables containing encryption keys), but it can often provide quite good security against the casual snooper, or against someone compromising the database file on the operating system.

As a trivial example, suppose you create a table (EMP) that contains employee data. You want to encrypt each employee's Social Security Number (one of the columns). You could encrypt each employee's SSN using a key which is stored in a separate column. However, anyone with SELECT access on the entire table could retrieve the encryption key and decrypt the matching SSN.

While this encryption scheme seems easily defeatable, with a little more effort you can create a solution that is much harder to break. For example, you could encrypt the SSN using a technique that performs some additional data transformation on the employee_number before using it to encrypt the SSN, something as simple as XORing the employee_number with the employee's birthdate, for example.

As additional protection, a PL/SQL package body performing encryption can be "wrapped," (using the wrap utility) which obfuscates the code so that the package body cannot be read. For example, putting the key into a PL/SQL package body and then wrapping it makes the package body — including the embedded key — unreadable to the DBA and others. A developer could wrap a package body called KEYMANAGE as follows:

```
wrap iname=/mydir/keymanage.sql
```

A developer can then have a function in the package call the DBMS_OBFUSCATION_TOOLKIT with the key contained in the wrapped package.

While wrapping is not unbreakable, it makes it harder for a snooper to get the key. To make it even harder, the key could be split up in the package and then have the procedure re-assemble it prior to use. Even in cases where a different key is supplied for each encrypted data value, so that the value of the key is not embedded within a package, wrapping the package that performs key management (i.e. data transformation or padding) is recommended. Additional information about the Wrap Utility is available in the PL/SQL User's Guide and Reference.

An alternative would be to have a separate table in which to store the encryption key, and envelope the call to the keys table with a procedure. The key table can be joined to the data table using a primary key-foreign key relationship; for example, EMPLOYEE_NUMBER is the primary key in the EMPLOYEES table, that stores employee information and the encrypted SSN. EMPLOYEE_NUMBER is a foreign key to the SSN_KEYS table, that stores the encryption keys for each employee's SSN. The key stored in the SSN_KEYS table can also be transformed before use (i.e. through XORing), so the key itself is not stored unencrypted. The procedure itself should be wrapped, to hide the way in which keys are transformed before use.

The strengths of this approach are:

- users who have direct table access cannot see the sensitive data unencrypted, nor can they retrieve the keys to decrypt the data
- access to decrypted data can be controlled through a procedure that selects the (encrypted) data, retrieves the decryption key from the key table, and transforms it before it can be used to decrypt the data

- the data transformation algorithm is hidden from casual snooping by wrapping the procedure, which obfuscates the procedure code
- SELECT access to both the data table and the keys table does not guarantee that the user with this access can decrypt the data, because the key is transformed before use

The weakness in this approach is that a user who has SELECT access to both the key table and the data table, who can derive the key transformation algorithm, can break the encryption scheme.

The above approach is not bullet-proof, but it is good enough to protect against easy retrieval of sensitive information stored in clear (e.g. credit card numbers).

Storing keys in the operating system (e.g. in a flat file) is another option. Oracle9*i* allows you to make callouts from PL/SQL, which you could use to retrieve encryption keys. However, if you store keys in the operating system (O/S) and make callouts to it, then your data is only as secure as the protection on the O/S. If your primary security concern driving you to encrypt data stored in the database is that the database can be broken into from the operating system, then storing the keys in the operating system arguably makes it *easier* for a hacker to retrieve encrypted data than storing the keys in the database itself.

Having the user supply the key, assumes the user will be responsible with the key. Consider that 40% of help desk calls are from users who have forgotten passwords, and you can see the risks of having users manage encryption keys. In all likelihood, users will either forget an encryption key, or write the key down, which then creates a security weakness. If a user forgets an encryption key or leaves the company, then your data is unrecoverable.

If you do elect to have user-supplied or user-managed keys, then you need to make sure you are using network encryption so the key is not passed from client to server in the clear. You also must develop key archive mechanisms, which is also a difficult security problem. Arguably, key archives or 'backdoors' create the security weaknesses that encryption is attempting to address in the first place.

Changing Encryption Keys

Prudent security practice dictates that you periodically change encryption keys. For stored data, this requires periodically unencrypting the data, and reencrypting it with another well-chosen key. This would likely have to be done while the data is not being accessed, which creates another challenge, especially so for a web-based application encrypting credit card numbers, since you do not want to bring the entire application down while you switch encryption keys.

SUMMARY

The growth of e-business necessitates that some especially sensitive information be stored in encrypted form. While encryption cannot address all security threats, the selective encryption of stored data can add to the security of a well-implemented application that uses a well-configured Oracle9*i* database. Oracle9*i* provides native encryption capabilities that enable application developers to provide additional measures of data security through selective encryption of stored data.

Appendix A

ENCRYPTION EXAMPLE

Following is a sample PL/SQL program to encrypt data. Segments of the code are numbered and contain narrative text explaining portions of the code.

```
DECLARE
 input_string        VARCHAR2(16) := 'tigertigertigert';
 key_string          VARCHAR2(8)  := 'scottsco';

 encrypted_string            VARCHAR2(2048);
 decrypted_string            VARCHAR2(2048);
   error_in_input_buffer_length EXCEPTION;
   PRAGMA EXCEPTION_INIT(error_in_input_buffer_length, -28232);
   INPUT_BUFFER_LENGTH_ERR_MSG VARCHAR2(100) :=
    '*** DES INPUT BUFFER NOT A MULTIPLE OF 8 BYTES ***';

1. Test string data encryption and decryption--    The interface
for encrypting raw data is similar.

BEGIN
   dbms_output.put_line('> ========= BEGIN TEST =========');
   dbms_output.put_line('> Input String                        :
' ||
 input_string);
   BEGIN
      dbms_obfuscation_toolkit. input_string => input_string,
                key_string => key_string, encrypted_string =>
encrypted_string );
      dbms_output.put_line('> encrypted string             : '
||
encrypted_string);
      dbms_obfuscation_toolkit.DESDecrypt(input_string =>
encrypted_string,
                key => raw_key, decrypted_string =>
decrypted_string);
      dbms_output.put_line('> Decrypted output            : '
||
                     decrypted_string);
      dbms_output.put_line('>  ');
      if input_string =
                     decrypted_string THEN
         dbms_output.put_line('> DES Encryption and Decryption
successful');
      END if;
   EXCEPTION
      WHEN error_in_input_buffer_length THEN
             dbms_output.put_line('> ' ||
INPUT_BUFFER_LENGTH_ERR_MSG);
   END;
```

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
+1.650.506.7000
Fax +1.650.506.7200
http://www.oracle.com/