Implementing Data-at-Rest Encryption within the Oracle RDBMS

James Forgy
RDC Software, November 2009
www.relationalwizards.com

Abstract:

Security requirements for compliance standards such as CISP, HIPAA, and PCI have increasingly emphasized data-at-rest encryption as a last-line of defense against data-theft. Although the Oracle Corporation offers Transparent Data Encryption (TDE) as a part of their Advanced Security Option (ASO) – highly transparent database encryption can be accomplished on most Oracle database platforms by implementing a set of best practices around a security-based methodology to protect data.

1. Introduction to Relational Data-at-Rest Encryption

Data-at-rest encryption within a relational database presupposes two things:

1. A lower-level encryption is not being used below the database level.

It is quite possible to purchase either encryption at the operating system level or the disk storage level. With these vendor tools, relational databases will write data to storage devices that will be encrypted, thus encryption within a database file is superfluous.

2. Encryption does not violate relational standards.

Simple tools can encrypt individual relational database data files, yet without incorporating the relational implications of this strategy, situations such as partially encrypted tables may incur, since data in a relational database only *appears as a collection of tables* and may span dynamic operating system files and storage devices.

Once we accept these two concepts, relational database encryption can be defined as a set of operations on rows and columns of table data. All SQL commands on table data, including encryption, are operations on rows and/or columns of data:

Table Army	Name	Rank	Serial Number
Row 1	Joe Smith	Private	001
Row 2	Sam Sanchez	Major	002
Row 3	Sue Jones	General	002

Fig 1 – The table Army with columns Name, Rank, and Serial_Number.

Column-Level Encryption

Column-level encryption is most common, and an easier specification of relational database encryption to implement. When defining column-level encryption, all rows for a given database column are encrypted:

Table Army	Name	Rank	Serial Number
Row 1	Joe Smith	Private	001
Row 2	Sam Sanchez	Major	002
Row 3	Sue Jones	General	002

Fig 2 – The table Army with columns Name and Serial_Number encrypted.

Row-Level Encryption

Row-level encryption is more difficult to implement, because only a subset of rows for any given column are encrypted based on a set of rules. These rules must ultimately translate to SQL *where conditions* or if-else logic to restrict row encryption during an encryption operation:

Table Army	Name	Rank	Serial Number
Row 1	Joe Smith	Private	001
Row 2	Sam Sanchez	Major	002
Row 3	Sue Jones	General	002

Fig 3 – The table Army, Name and Serial_Number encrypted for rank of general only.

Row-level and column-level encryption are the two apparent results of all encryption operations defined on relational data. Encrypting database data using these relational definitions as a standard is not difficult, yet the implementation of encryption operations on a given data domain must be exact and carefully planned to insure security and data consistency.

In both of the above examples of table encryption, once the appropriate rows are marked for encryption, a SQL update statement will be used to convert regular data into encrypted data using a cipher of choice such as AES-256 or DES3.

After the actual process of encrypting data is complete, old unencrypted data may still exist in Oracle data files, marked as unused blocks; therefore in a highly secure environment, backing up the newly encrypted data and importing it to a fresh database installation on recently formatted storage devices is the safest practice to remove all references to encrypted data from a physical database server.

2. Cipher Key Generation

Data encryption standards such as AES and DES3 rely on a *key value* for the *cipher* to obfuscate data. For any encryption operation, ciphers must yield an *identity function* or encrypted data will be destroyed:

f(g(x)) = x *where x is relational data.*

Encryption functions are dependent on the key value or *pass-phrase* to transform data into safe unreadable data and to render it readable again. This *private key* dependency implies that key generation must yield near-random cipher keys to properly hide (randomize) different data sets within a relational database.

Aside from user supplied pass-phrases, encryption keys are usually generated by a pseudo-random hash function such as SHA-2. Because relational databases contain large amounts of data in very distinct patterns, such as ANSI characters, random keys are important to thwart attempts at cracking an encryption algorithm.

3. Key Management Strategies

Once data in a relational database is encrypted using a cipher and a key, that key must be stored somewhere for successful read operations (decryption) or write operations (encryption) to encrypted database tables.

There are three ways to store keys when encrypting relational data:

Within the Database

This is the most common and fastest way to retrieve keys when performing SQL operations on encrypted data. Because a key value itself is the only way to render encrypted data useful, it must also be encrypted when stored in the database for security purposes.

Luckily, these keys have no intrinsic value, so both traditional encryption methods such as AES or hashing methods can be used to store cipher keys. When using a hashing algorithm to store a cipher key, the key itself is not stored in the database, but a seed that will become the cipher key at runtime:

Seed => Hash Function => Key Value

The advantage of storing cipher keys in a relational database using a hashing algorithm is that it is a one-way function, thus the encrypted data itself does not give you information to match the original seed in the database.

Outside the Database

In this scenario, cipher keys are not stored within the relational database. For instance, highly secure key values may be stored on removable flash memory and kept in a safe. Before any operations on relational data can occur, the safe must be opened, the removable hardware mounted by the file system and the key values read into memory as needed by the process executing the cipher.

Nowhere

Key values do not need to be stored on any electronic medium. A user can supply a pass-phrase and simply remember it or write it down on a napkin. Once the data is encrypted with the supplied phrase, the same phrase will be needed to render the data useful again for any encryption or decryption operation. If the password is forgotten or lost, the data will remain useless forever.

4. Transparent Operations

As evidenced, sound key management techniques are crucial for data-at-rest encryption to be secure and consistent. Likewise, transparency is critical for data-at-rest encryption to be cost-effective. Transparency in relational database encryption is defined as the degree to which encrypted table data *can appear,* in both structure and content, as identical to original data.

In a situation of perfect transparency, applications that are authenticated would be working off the identical schema and table data as they would be if data was unencrypted. Migration costs would be negligible. On the other hand, zero transparency implies all data structures change after encryption—applications would have to be drastically modified to view encrypted data, a costly enterprise.

Transparent encryption can be purchased as a part of Oracle's Advanced Security Option (ASO) and customers whose needs or budget do not necessitate this expenditure can build or buy third-party tools, such as the Encryption Wizard for Oracle, that will also deliver very high transparency.

Highly transparent encryption in Oracle can be implemented through the use of *views* and *database triggers* on those views. These views will display encrypted data and appear as the original tables to applications. Because these views decrypt data to make it readable at runtime, they are referred to as *decrypted views*. They are created for each encrypted table *after* table data has been encrypted. A public synonym can then be created, assigning the original table name to the new decrypted view that will serve applications their data.

Consider the table Army as shown in *figure 1* earlier. After two columns of this table are encrypted, a view can be created *v_army* to allow users to view the original data:

```
Create View V_Army
As Select
DECRYPT(Name, Column Identifier),
Rank,
DECRYPT(Serial_Number, Column Identifier)
From Army;
```

Notice embedded within the *create view* SQL is a user-defined function *decrypt* containing the cipher. This function will decrypt two of the encrypted columns in the view, thus creating data transparency for any application querying the view. The *Column Identifier* field is a numerical constant used so the *decrypt* function knows which column's key value to retrieve.

Depending on how key values are stored, the function *decrypt* will retrieve the appropriate key from the database or maybe a mounted storage device. After this, *decrypt* will call a traditional cipher mechanism, such as Oracle's DBMS_Crypto utility or a Java library like GNU Crypto, and then return the result to the view.

Write transparency is accomplished through the use of a *row-level instead-of trigger* referencing a decrypted view. When a user inserts new unencrypted data into the view *v_army*, the following PL/SQL logic will encrypt the two appropriate columns, a necessary operation to add encrypted data to a table:

```
If Inserting Then
        Insert Into Army Values
        (
                ENCRYPT(:new.name, Column_Identifier)
                ENCRYPT(:new.serial_number, Column_Identifier)
                :new.Rank
        )
End If;
```

Likewise, update and delete conditions must also be coded within this trigger to provide full read/write transparency for the decrypted view that the above instead-of trigger references.

5. Row-Level Transparency

Row-level encryption provides more of a challenge for transparent SQL operations within a database. For instance, the above trigger would have to be modified so as to only encrypt a subset of rows within a given table – either through the use of if-else logic or *where conditions* added to the SQL.

In the previous trigger, using standard ANSI SQL, two insert statements would be utilized for row-level encryption — one insert without the *encrypt* function for rows not encrypted, and another insert statement with *encrypt* embedded.

These same row-level rules could also be implemented within the *encrypt* function itself to avoid repetitive SQL statements. One shortcut is the *decode statement* in Oracle SQL to implement row-level encryption logic:

Decode(:new.rank, 'General', ENCRYPT(:new.name, *Column Identifier*), :new.name)

This decode statement would be embedded in an insert or update SQL statement, thus only officers at the rank of general are encrypted, as depicted visually in *figure 3* previously.

6. Runtime Authentication

Even with best practices in key management and encrypted data transparency, relational database encryption will not be secure without runtime authentication. A relational database already contains authentication in the form of a user name and password, yet absent any additional authentication for encrypted data, simply stealing someone's password with decryption/encryption privileges is all that is needed for data theft — essentially, leaving an encrypted database not much more secure than it was before encryption.

Secure encryption must authenticate relational database users at runtime based on the security concerns of the underlying schema, table or column. Ironically, the situation where cipher keys are stored *nowhere* and memorized, runtime authentication is accomplished by definition. For all other forms of key management there are two basic strategies that can be used to authenticate database users against encrypted objects:

1. Require users to provide the pass-phrase used to generate the encryption key for a given database schema, table or column.

2. Require users to provide a pass-phrase, not related to the key, before operations on a given database schema, table or column.

The advantage of the first method is that the user must supply the actual key used to encrypt/decrypt data – without this key, data cannot be accessed. Therefore data-theft requires knowledge of the key value itself or its derivative and not simply the ability to steal a password.

The disadvantage of requiring users to enter the actual cipher key, aside from key length considerations, is that this pass-phrase cannot change without decrypting and encrypting the entire data domain with a new key, *re-keying*. Although best security practices dictate periodic password and cipher key changes, re-keying becomes more costly proportionate to the size of the data that needs to be re-keyed and the frequency with which passwords are changed.

7. Row-Level Key Generation and Management

In the art of cryptography and data theft, deciphering encrypted data, cracking its key, grows easier as the sample size of encrypted data grows. A single row of encrypted data is not large enough to find a pattern that will lead to deciphering a key — a row is just a few random bytes; yet one million rows of data, is actually less secure and easier to find patterns within the data. Because relational databases store large amounts of repetitive character set data, this makes them inherently more vulnerable to cryptography cracks than most traditional operating system files.

One way around this inherent weakness is using a different key value for every row of data. This creates a high-degree of randomness and forces many key values to be cracked – an almost impossible task without knowing the seed used to generate the key.

The drawback of this method is that the key must be based on either a portion of the data in each row or another unique identifier that distinguishes the row of data. This dependency also implies that if this identifying data changes, the key changes; suddenly encrypted data cannot be deciphered and is rendered useless or worse yet — lost.

Row-level key management is based on this important rule:

The data identifier used to build each row's keys must not change, unless the encrypted data is re-keyed with the new resultant key.

Therefore, to generate a distinct cipher key for each row of data we must pick a unique database key of the table or another unique *row identifier* that does not change. Using the *primary key* of a table as the seed can be done without much effort — if the column(s) of that primary key never change:

```
ENCRYPT(:new.name, :new.serial_number)
```

In this example, the unique column *serial_number* is used to generate a key-value to encrypt the column *name* for all given rows. Because the *serial_number* does not change, it is an ideal row-level key. Yet if a SQL command does update the primary key, the data is lost, unless a row-level database trigger re-keys the encrypted data dependent on the new key.

Two ways around the update restriction of data-driven keys is to utilize an internal identifier, such as Oracle's *rowid*, to generate a cipher key. A rowid is a unique pointer that the Oracle database uses to store data offsets requested by SQL operations. The rowid value is accessible to a database trigger and can be employed to generate a cipher's key:

```
ENCRYPT(:new.name, :new.rowid)
```

The trade-off in utilizing a relational row identifier is that in some database implementations this value can change. For instance, if data is recovered from a backup after a disk failure — the rowid *might* change. Changes can also occur for simpler operations such as defragmentation performed against relational data. Therefore, unless the exact behavior of the rowid and the types of database administration performed against the data are known, it is best to avoid using the rowid to generate cipher keys.

8. Bit-mapped Indexes for Performance and Application Transparency

A traditional binary index in a relational database is used to speed up SQL operations. Yet with encrypted data sets, a traditional binary index that matches values in a SQL where condition to physical rows, by definition, creates a security problem, because unencrypted values must be stored in the index's *b-tree leaf nodes*:

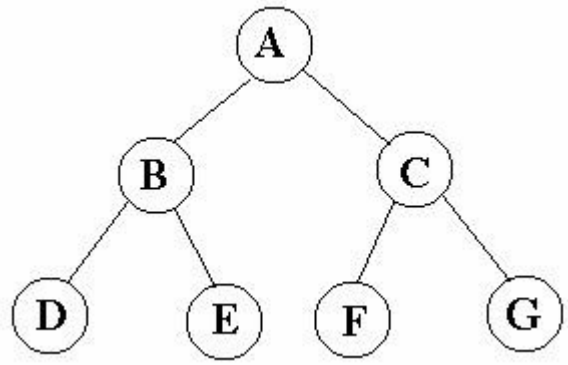

Figure 4 – Abstract representation of a binary tree index with nodes D through G as leaf nodes

In this abstract representation of a database index, nodes D through G are *leaf nodes* and thereby must store the unencrypted column value and a matching row identifier to optimize SQL *where conditions* issued by database applications expecting transparency.

For instance, an authenticated user might issue this SQL to promote a soldier:

```
Update V_Army Set Rank = 'Sergeant' Where Name = 'Joe Smith';
```

By definition, a binary index used to optimize this SQL request must contain the original unencrypted value *Joe Smith* in one of its leaf nodes. Using a traditional function index to accomplish this will leave original table data exposed in the index tablespace. Even though data stored within an index tablespace is less accessible, it is still unencrypted and can be attained by a hacker with access to data files.

Compounding this security issue would be a composite index containing both *name* and *serial_number.* In this case data theft would be trivial once the underlying data files of the index tablespace were stolen – rendering encryption almost useless, since index data files would contain matching values of sensitive columns.

One way around this paradox is the use of bit-mapped function indexes. A bitmap index stores a hash representation of data in the index tablespace instead of the data itself. Because this hash value is designed to yield row identifier information for SQL operations, data theft of bit-mapped data is more difficult – ironically maybe even more so than cracking traditional ciphers such as DES3.

When using a bit-mapped function index to replace a traditional b-tree index for an encrypted column, both the *decrypt* function and the bitmap specification must be referenced in the create index statement:

```
Create Bitmap Index MyIndex on Army(DECRYPT(Name, Column Identifier))
```

This SQL DDL operation instructs the Oracle database to create a bit-mapped function index on the encrypted column *name.* The function used is *decrypt,* which will decrypt values to their original form and allow the Oracle kernel to then generate a bitmap representation for faster SQL operations:

```
Encrypted Data => DECRYPT=> Bitmap Hash => Value in Index Tablespace.
```

Using bit-mapped indexes does preclude the use of the rowid to provide a row-level key. This statement is not valid:

```
Create Bitmap Index MyIndex on Army(DECRYPT(Name, RowID))
```

Yet the use of a unique database key to generate row-level cipher keys is allowed:

```
Create Bitmap Index MyIndex on Army(DECRYPT(Name, Serial_Number))
```

In this last example, the *serial_number* column is once again used in the *decrypt* function to generate a row-level cipher key for the *name* column. Oracle will store the index data of the column in bitmap fashion. This insures both security and high-performance SQL operations on encrypted data.

There is one problem that bitmap indexes do not solve and that is database sorting algorithms for a query such as:

```
Select * from V_Army Order by Name, Serial_Number;
```

In cases such as these, Oracle *may* need to sort the unencrypted Name and Serial Number in a temporary tablespace located on a physical disk. Therefore, best security practices for encrypted databases dictate periodic recreation of the Oracle database *tempfiles,* a quick process if they are *managed locally.*

9. Conclusion

Encrypted data-at-rest is the new standard for secure relational database environments. This requirement, when integrated with traditional database applications, poses a series of security and performance choices that need to be addressed at the outset of any encryption project.

For customers whose business or project needs do not require Oracle's Advanced Security Option, using best security practices and the simple data structures discussed in this paper will afford database environments a robust and secure encryption solution. Third-party tools, such as the Encryption Wizard for Oracle, are also available to automate this cost-effective paradigm of transparent data encryption.

As evidenced, the easiest phase of a database encryption project is encrypting the data itself. The more difficult tasks are key management, transparency, performance tuning, and user authentication. Planning for these tasks before migrating to encrypted table data will increase the chances of a project's success and, more importantly, its contribution to enhanced data security.

References

1. Pete Finnigan: "*The Right Method to Secure an Oracle Database*". October 2009, www.petefinnigan.com/Oracle_Security_OWASP_Leeds_2009.pdf

2. RDC Software: "*The Encryption Wizard for Oracle User Manual."* September 2009, www.relationalwizards.com/ew_docs/EncryptionWizardJava.pdf

3, Arup Nanda: "*Protect from Prying Eyes: Encryption in Oracle 10g*." January 2006, www.dbazine.com/olc/olc-articles/nanda11

4. Oracle Corporation: "*Database Encryption in Oracle9i*." 2001, hwww.cgisecurity.com/database/oracle/pdf/f5crypt.pdf .

5. Jakub Wartak: "*SHA1, SHA256, SHA512 in Oracle for free without using DBMS_CRYPTO*." June 2009, http://vnull.pcnet.com.pl/blog/?p=124 .

6. E. F. Codd: "*A Relational Model of Data for Large Shared Data Banks*." June 1970, www.seas.upenn.edu/~zives/03f/cis550/codd.pdf

7. Oracle Corporation: "*Oracle Advanced Security Transparent Data Encryption Best Practices*." August 2009, www.oracle.com/technology/deploy/security/database-security/pdf/twp_transparent-data-encryption_bestpractices.pdf

8. Oracle Corporation: *"Bitmap Index vs. B-Tree Index: Which and When?"* www.oracle.com/technology/pub/articles/sharma_indexes.html

9. Oracle Corporation: "*Oracle Advanced Security with Oracle Database 11g R2."* September 2009, www.oracle.com/technology/deploy/security/database-security/pdf/owp-security-advanced-security-11gr2.pdf

10. Copacobana: "*A Codebreaker for DES and other Ciphers*." May 2008, http://www.copacobana.org/