USING DATATYPE-PRESERVING ENCRYPTION TO ENHANCE DATA WAREHOUSE SECURITY

Michael Brightwell
FM Software, Inc.
621 17th Street, Suite 825
Denver, Colorado 80293
telephone: 303-298-8262
email: mbrightw@fmsoftware.com

Harry E. Smith
Quest Database Consulting, Inc.
5600 South Quebec Street, Suite 310-D
Greenwood Village, CO 80111
telephone: 303-771-2246
email: hsmith@qdbc.com

Abstract

A well-designed data warehouse invariably contains information which must be considered extremely sensitive and proprietary. Protection of this information, as important as it is, is too often complicated by the presence of heterogeneous computing environments, organizational politics, difficulties in controlling data distribution, and lax attitudes towards information security. We present a method of information protection, based on an encryption scheme which preserves the datatype of the plaintext source. We believe that this method is particularly well-suited for complex data warehouse environments.

Introduction

Data warehouse technology has, in recent years, provided corporate executives and business planners with extraordinarily powerful decision-support tools. Data warehouses can tell us *which* products to manufacture, *where* to locate factories and *how* to gain market share. They can be used to answer questions which weren't even contemplated when the data warehouse was built. Most remarkable of all—these feats can be accomplished using data extracted from existing operational systems.

It has become increasingly apparent, however, that the data warehouse is an inviting target for snoopers. Imagine that you are a disgruntled employee, social activist, or industrial espionage agent who manages to gain access to an organization's computer system. Now, what would you like to find?

- Data grouped into subject data areas so that you can quickly find the items of interest.
- Accurate and complete information that has been painstakingly reviewed for correctness.
- Time-indexed data so trends can be easily identified.
- Summarized information with the ability to drill down for details.

In short, you would be looking for a data warehouse!

Conceptually, the data warehouse process consists of three simple steps:

1) Extract data from the operational system,
2) Load the extracted data into data warehouse tables, and
3) Query the data warehouse to obtain decision-support information.

Data is at risk during each of these phases. Several factors render data warehouses particularly susceptible to attack:

1) Extracted data is frequently transmitted of over insecure communication lines.
2) Extracted data is stored on a variety of computer systems and removable media which may have only minimal security.
3) The extraction process produces intermediate files and load files which contain sensitive information, but may not be well-protected.
4) Maintaining proper security attributes for the data warehouse tables is extremely time consuming in the face of constant organizational change.
5) Users often retrieve data from the data warehouse and create a "data mart," leading to widely distributed copies of sensitive data.
6) Sound security practice are often undermined because the data warehouse development effort is a "high visibility" project with a tight schedule. (It is a brave developer, indeed, who is prepared to tell senior management that the

delivery of their long-awaited decision support tool will be delayed in order to investigate problems which *might* happen in the future.)

Furthermore - given that research into the unique aspects of data warehouse security is still in the early stages - additional vulnerabilities will certainly be identified in the future.[1]

Problem Statement

When we decided to develop a cryptographic approach to data warehouse security which could be applied in the complex, heterogeneous environments encountered in the business world, we identified certain critical objectives:

1) The approach should work with any combination of commonly used relational databases. (This rules out requiring binary data storage or other database-dependent features.)
2) It must function on multiple hardware platforms and operating systems.
3) It must correctly encrypt and decrypt data on machines with different character sets (e.g., ASCII and EBCDIC).
4) The strength of the encryption algorithm should be comparable to widely-used, state-of-the-art technology.
5) Adding encryption to an existing database should require no changes to the structure of the database. (Neither should any application changes be required to access non-encrypted fields.)
6) Encryption should occur as early as possible in the extraction process and decryption should occur at the last possible moment.
7) It could not be dependent on a particular programming language.
8) It should be "fail safe." (Any likely failure mode should be such that that access to the data is *denied*.)

These requirements, based on our business requirements, constituted a formidable challenge. During the course of our research, however, it became apparent that the tractability of the problem could be improved significantly if we could find a way to preserve the original datatype across the encryption and decryption transformations.

Proposed Solution

Ciphertext (data in encrypted form) bears roughly the same resemblance to plaintext (data in its original form) as a hamburger does to a T-bone steak. A social security number, encrypted using the DES encryption algorithm, not only does not resemble a social security number but will likely not contain any numbers at all. A database field which was defined to hold a nine-character social security number (eleven, if you want to include the hyphens) would not be able to store the DES-encrypted version of the data. A Visual Basic program would not read it. A graphical interface would not display it. There would be nothing that you could do with the encrypted social security number unless you had made extensive provisions for changes in data format throughout your application and physical database design.

Basic Datatype Preservation

Our method reduces the need for changes to database structures and applications by preserving the datatype of the encrypted field. Datatype preservation simply means that each ciphertext field is as valid as the plaintext field it replaces. The key to our approach is defining an appropriate alphabet of valid characters and performing all operations within the constraints of the defined alphabet.

Each different datatype requires a judicious choice of alphabet. An alphabet consisting of numeric digits ("0123456789") could be used to encrypt most number data, such as social security numbers (e.g. 123-45-6789). (The dashes, not included the chosen alphabet, are copied unchanged to the corresponding positions in the ciphertext output.) Other alphabets, such as all printable ASCII characters, all characters shared by ASCII and EBCDIC, or all hexadecimal digits can be used to encode a variety of common datatypes.

---

[1] Credit for early recognition of these special data warehouse security problems goes to Bill Inmon, the "father of the data warehouse." Bill also gets credit for the concept of encrypting the data during extraction and decrypting it just before it is presented to the user to provide a "safety net" in case of compromise of one of the standard security mechanisms. See, for example, his presentation at the April 21-25, 1996 "Data Warehouse and Decision Support Systems 96'" conference in Arlington, VA (Barnett Data Systems, 19 Oracle Way North, Rockville, MD 20854)

The Approach

The first processing step involves replacing each plaintext character in the string by an integer that represents its position, or index, within the chosen alphabet. This number is between zero and one less than the total number of characters in the alphabet. If a plaintext character is not in the valid alphabet, it is copied to the output and removed from the string to be encrypted.

Figure 1 includes a worked example of the basic algorithm.:

Example:

plaintext = "hello"

alphabet = "abcdefghijklm
nopqrstuvwxyz"

Step 1: Assign Index Values

index values =7, 4, 11, 11, 14

Step 2: Add Position Sensitive Offsets

offsets = 10, 5, 18, 25, 4

new index values = 17, 9, 3, 10, 18

Step 3: Shuffle the Index Value String

shuffled values = 3, 18, 17, 10, 9

Step 4: Convert Back to Desired Datatype

ciphertext = "dsrkj"

Figure 1: Basic Processing Example

After the alphabet index values have been assigned, we add a varying integer "offset" to each. We use modular addition to ensure that we generate only valid characters (i.e. characters which are contained in the alphabet). Remember that "modular" addition means adding two numbers and then determining the remainder after division by a constant "modulus" value. In the example above the alphabet size is 26 so, for example, 18 + 11 (mod 26) = 3. The actual offset values are generated based on a portion of the key being used to encrypt the data. This step ensures that long series of identical characters (such as 20 blanks at the end of a character field) will not encrypt identically.

After adding the offsets, the entire string is shuffled. The shuffling method varies according to a permutation-invariant property of the index values, such as a sum or exclusive-or, of all values.[2] The shuffling step helps to ensure that plaintexts with common prefixes or suffixes do not produce ciphertext with common prefixes or suffixes.

Once the encoding process is complete, each index value is mapped to the appropriate character in the alphabet.

To recover the plaintext from the ciphertext, one replaces the ciphertext characters by their alphabet index values, "unshuffles" the string, regenerates the offset values, subtracts modularly on an integer-by-integer basis and substitutes the appropriate alphabet character.

Two enhancements to the above algorithms may be used to deal with certain data-specific situations:

First, in order to ensure that the encoded values of two single character strings with adjacent characters are not sequential (for example, we would not want "b" to encrypt as "y" whenever "a" encrypts as "x"), the alphabet itself can be shuffled based on a portion of the encryption key.

Second, in order to inhibit guesses based on encrypted character permutations, we can "ripple" the data from left to right and from right to left. This is done by hashing the key into a "starter-digit" and adding adjacent values pairwise. For example, the string of index values "1, 2, 3" might be rippled into "23, 5, 40" as follows (assuming a 55 character alphabet):

starter value = 72 (obtained by hashing the encryption key)

adding left to right:

72 + 1 (mod 55) = 18
18 + 2 (mod 55) = 20
20 + 3 (mod 55) = 23

adding right to left:

23 + 72 (mod 55) = 40
20 + 40 (mod 55) = 5
18 + 5 (mod 55) = 23

Applying this same method to the permutation "3, 2, 1," on the other hand, ripples it to "27, 7, 40" and the fact that the two strings contain the same characters is disguised.

---

[2] A variety of techniques could be used to generated the offsets and shuffling pattern, including the use of pseudo-random number generators.

Enhanced Encryption

While the encoding scheme presented above is sufficient to deter casual attacks ("keeping your sister out" as Bruce Schneier would put it[3]), more substantial protection is required to protect sensitive data in the data warehouse. The approach described above can be combined with well-known encryption algorithms, such as DES or IDEA, to significantly increase the attacker's burden. The basic idea is to use an established algorithm of known strength to produce the "offset" values.

The DES algorithm takes as input a 64-bit input block and a 64-bit key (56 key bits and 8 parity bits) and uses these two values to produce a 64-bit output. The ciphertext output can be decrypted using the same key. For all practical purposes, the only way to break the scheme is by an exhaustive search of the keyspace.

DES, like any block cipher, can be operated as a stream cipher in "cipher-feedback" mode. We use this mode to encrypt one index value at a time. At the end of each encryption pass, we also shift the plaintext data into the DES input register. This process is illustrated as follows:

Let the alphabet index values of the n-character, plaintext input string be represented by

$i_1\, i_2\, i_3\, i_4\, i_5 \ldots i_n$

Let the 64-bit DES initial value required by cipher-feedback mode be constructed based on a portion of the encryption key

$H(K) = a_1\, a_2\, a_3\, a_4\, a_5\, a_6\, a_7\, a_8 = A$

where each subscripted "a" value represents an 8-bit number ("0" to "255"). Let the output of the DES algorithm, using a key of "K" and an input of "A," be represented by

$E_k(a_1\, a_2\, a_3\, a_4\, a_5\, a_6\, a_7\, a_8) =$

$b_1\, b_2\, b_3\, b_4\, b_5\, b_6\, b_7\, b_8.$

The first transformed index value is the modular sum

$z_1 = b_8 + i_1 \pmod{l}$. Where "l" represents the alphabet length.

At this point, a new DES input value, A, is constructed as

$A^2 = b_2\, b_3\, b_4\, b_5\, b_6\, b_7\, b_8\, i_1$

and a new DES output is obtained

$E_K(b_2\, b_3\, b_4\, b_5\, b_6\, b_7\, b_8\, i_1) =$

$c_1\, c_2\, c_3\, c_4\, c_5\, c_6\, c_7\, c_8.$

The second transformed index value is the modular sum

$z_2 = c_8 + i_2 \pmod{l}$.

Note that the use of an addition operator is required, instead of the usual exclusive-or operator, is required to ensure that the datatype is preserved.

After n such steps, during each of which a single input index value is transformed, we have an encrypted index-value string

$Z = z_1\, z_2\, z_3\, z_4\, z_5 \ldots z_n$

We claim that recovering the string, $i_1\, i_2\, i_3\, i_4\, i_5 \ldots i_n$, from the transformed string, $z_1\, z_2\, z_3\, z_4\, z_5 \ldots z_n$, without knowledge of the key, K, is as difficult as breaking the DES algorithm itself.

When using cipher-feedback mode, DES decryption, per se, is never invoked. Reversing the transformation is done by subtracting the low order DES output from the transformed index value.

Below is a summary of the algorithm.

Setup tasks:

1) Choose an encryption key with enough bits for the encryption algorithm key, encryption algorithm initial value and any basic processing stages.
2) Choose a suitable alphabet to support the datatype of the data to be encrypted.
3) Shuffle the alphabet according to a scheme based on the key.

For each encrypted field:

4) Scan the input buffer for characters which are not included in the chosen alphabet. Move all invalid characters unchanged to their corresponding positions in the ciphertext output buffer.
5) Move the index values of all valid characters to adjacent positions in a work buffer.
6) Add position-sensitive offsets according to a key-dependent scheme.
7) Shuffle the work buffer positions according to a data-dependent scheme.
8) "Ripple" the work buffer by calculating a key-based starter number and modularly adding pairwise from left to right then from right to left.

---

[3] Schneier, Bruce, Applied Cryptography, New York: John Wiley, 1996

9) Set the cipher-feedback initial value using the chosen key.

10) Calculate the modular sum of the first work buffer position and the lowest-order DES output byte. Store this value in a second work buffer.

11) Obtain a new DES initial value by moving the DES output to the input, shifted one byte to the left, and shifting the work buffer value into the lowest-order position.

12) Repeat steps 9 through 11 using successive work buffer index values until all of the data is transformed.

13) Replace the transformed index values by their corresponding character equivalents and store them in the open ciphertext positions.

Decryption is accomplished by performing the inverse of each transformation in the reverse order.

Implementation Issues and Usage Constraints

Perhaps the most important *caveat* for anyone who wishes to implement our proposed encryption scheme is to guard against possible misinterpretation of encrypted data. Scrambled text fields such as names and addresses are not likely to be mistaken for real information, but numeric fields may contain quite plausible values. A legitimate user who, through some administrative oversight, is erroneously presented with encrypted data may not recognize it as such and make bad decisions as a result.

One approach to this shortcoming may be to include code in the query tool to fill in encrypted fields with a default value whenever the user has not been authenticated. Another approach may be simply to restrict the application of the technique to text fields. A revenue field may be quite useless without the corresponding product data or sales region information.

Another restriction on the use of this technique is that decryption must be performed before aggregate functions, such as minimum, maximum, sum, and average, are applied. This is not a serious inconvenience in the data warehousing environment because precomputed summary tables are usually available.

One must also bear in mind that this encryption scheme is *consistent* in that the same plaintext always results in the same ciphertext. This has both positive and negative implications. On the positive side, the consistency of the encrypted data allows for relational joins and *blind keys* (described later). On the other hand, consistent encryption exposes the data to the possibility of a statistical attack. If an attacker knows the relative frequency of specific data items, such as medical tests, he can deduce the corresponding encrypted values. This kind of attack can be stymied by using a value from another field (the table's primary key, for example) to modify the encryption key. This would, of course, preclude the use of this data in relational join predicates.

Coexistence with Other Security Controls

We do not propose datatype-preserving encryption as the ultimate solution to all data warehouse security concerns. It is presented, rather, as one of several mechanisms to be employed in a more comprehensive security strategy. Specifically, we see our technique as a *containment* device which limits potential damage in the event of a successful bypass of other security controls.

In general, there are at least five categories of security controls:

Prevention. Preventative measures include anything which can be done to prevent an attack or to keep it from succeeding. This includes strengthening vulnerabilities and providing disincentives to the attacker.

Detection. Detective measures include anything which alerts the support staff to the fact that an attack is in progress or has been recently attempted.

Containment. Containment measures include anything which can serve to limit the damage of a successful attack.

Recovery. Recovery measures include anything which is done to restore normal operation and user access after an unscheduled interruption.

Investigation. Investigative measures include anything which is done to identify a malefactor and collect evidence which will be used in a disciplinary process or criminal prosecution.

A good data warehouse security plan will include multiple countermeasures for each identified threat—ideally, at least one from each of these categories.

Applications to Other Areas

In addition to data warehouse security, there may be several other areas in which

this technique may prove useful, such as providing an additional check on data integrity. By adding a check character to the beginning of each plaintext field (easily automated during database load) any alteration would be immediately obvious during the decryption process. The decryption routine could be modified to perform this check and return an error code is tampering is suspected.

Another possible application is in the use of *blind keys*. In certain situations, one needs to know that two quantities are equal without actually knowing the quantities themselves. One may wish to match bank account numbers from multiple sources, for example, in credit check applications but not use the actual numbers themselves because of the potential for fraudulent activity.

It may also be possible to control access to commercially available data through the use of this technique. A master database could be distributed to subscribers with individually licensed components encrypted using different keys. Access to the individual components could be made available by distributing keys following payment of the proper license fees.

Conclusion

Progress in the field of cryptology is based on the practice of making the algorithms public and inviting interested parties to find the flaws. It is in this spirit that our method is presented. We welcome comments, criticisms and suggestions for improvement from readers. Please feel free to contact either of the authors at the addresses provided.