Chapter 16
Security

While database integrity means protecting it against authorized users, database security means protecting data against unauthorized users.

Integrity is part of the DB foundation, while security is secondary, which makes a better product.

There are numerous aspects to the security problem: legal, social, and ethical aspects; physical controls; policy questions; operational problems, Hardware controls; operating system support; and issues that are the specific concern of the database itself.
Main approaches

There are two broad approaches to database security: discretionary and mandatory control.

In the former case, a given user will typically have different access rights on different objects, but there are very few limitations regarding which users can have which rights on which objects.

In the latter case, each object is labeled with a certain classification level, and each user is given a certain clearance level. A given data object can then be accessed only by users with the appropriate clearance. Thus, it is hierarchical in its nature.
Discretionary access control

Usually, it is authorities, but not constraints, that are defined. For example,

\texttt{AUTHORITY SA3}  
\texttt{GRANT RETRIEVE (S\#, SNAME, CITY), DELETE}  
\texttt{ON S}  
\texttt{TO Jim, Fred, Mary;}

In general, authority is registered under a \textit{name} such as SA3. One or more \textit{privileges} are specified via the \texttt{GRANT} clause. The \textit{table} to which the authority applies is specified with the \texttt{ON} clause. Finally, one or more \textit{users} who will be granted the specified privileges over the specified table is specified via the \texttt{TO} clause.
When some user attempts to apply certain unauthorized operations on an object, the obvious thing to do is to reject the attempt. Sometimes, some other reactions might be more appropriate. For example, simply terminate the program, lock up the user’s keyboard, or even record such attempts in a special log file to permit further analysis of attempted security breaches.

It is also easy to drop an authority, e.g.,

DROP AUTHORITY SA3;
Examples

The following allows users Jacques, Anne, and Charley to see a "vertical subset", namely, a projection, of P. Thus, a value-independent authority.

AUTHORITY EX1

GRANT RETRIEVE (P#, PNAME, WEIGHT)
ON P
TO Jacques, Anne, Charley;

Below is an example of value-dependent authority applied on a view.

AUTHORITY EX2

GRANT RETRIEVE, UPDATE(SNAME, STATUS), DELETE
ON LS
TO Dan, Misha;
The following is another value-dependent authority, which only allows Giovanni to retrieve supplier information, who supply some part stored in Rome.

VAR SSPPR VIEW (S JOIN SP JOIN (P WHERE CITY=‘Rome’) {P#} {ALL BUT P#,QTY};

AUTHORITY EX3
GRANT RETRIEVE
ON SSPPR
TO Giovanni;

Similarly, we can add in some context controls to specify context-dependent authority.

AUTHORITY EX5
GRANT RETRIEVE, UPDATE(STATUS)
ON S
WHEN DAY() IN (‘Mon’,‘Tue’,‘Wed’)
   AND NOW()>=TIME‘09:00:00’
   AND NOW()<=TIME‘17:00:00’
TO Purchasing;
SQL facilities

The current SQL standard supports discretionary access control only. Two independent SQL features are involved: the view mechanism can be used to hide sensitive data from unauthorized users, and the authorization subsystem allows users having specific privileges grant those access rights to other users, and later, revoke them, if desired. Below defines the data over which authorization to be granted:

CREATE VIEW LS AS
  SELECT S.S#, S.SNAME, S.STATUS, S.CITY
  FROM S
  WHERE S.CITY = 'London';

GRANT SELECT, UPDATE(SNAME, STATUS), DELETE
  ON LS TO Dan, Misha;

REVOKE SELECT ON S FROM Jacques RESTRICT;
Request modification

Now, we discuss the query language QUEL, which adopt another approach to the problem. Any QUEL request is automatically modified before execution so that it can’t possibly violate any specified security constraint. E.g., assume user U is allowed to retrieve parts stored in London only:

```
DEFINE REMIT RETRIEVE ON P TO U
    WHERE P.CITY="London"
```

Now assume U issues the request:

```
RETRIEVE(P.P#,P.WEIGHT)
    WHERE P.COLOR="Red";
```

The system will automatically rewrite the request to the following:

```
RETRIEVE(P.P#,P.WEIGHT)
    WHERE P.COLOR="Red"
    AND P.CITY="London";
```
This request modification process is the same as the technique used to implement views. Also, it is relatively efficient, since the associated overhead occurs at compile time instead of run time.

Compared to the SQL approach, this is more flexible to deal with the case when a user needs various privileges over different portions of the same table. For example, it will be awkward to specify the respective rights that allow Bob to check out everything in P, but only update those red parts. (?)

```
CREATE VIEW RP AS
    SELECT P.P#, P.SNAME, P.COLOR, P.WEIGHT, P.CITY
    FROM P
    WHERE P.COLOR='Red';

GRANT SELECT ON P TO Bob;

GRANT UPDATE ON RP TO Bob;
```
Two sides of the same coin

It will be much easier to do the same, following the just discussed alternative approach.

```
DEFINE REMIT RETRIEVE ON P TO Bob;
DEFINE REMIT UPDATE ON P TO Bob
WHERE COLOR='Red';
```

But, not all security constraints can be handled in this simple way. For example, if Bob is not allowed to access a table at all, there is no simple “modified” form of the query that says that “P does not exist.” A specific message must be printed in this case.

**Homework:** Exercises 17.1 and 17.8.
Audit trails

In case the system is broken into, an *audit trail* becomes a necessity. It can be used to determine what has been going on, to verify that matters are under control, or to help pinpoint the infiltrator.

An audit trail is essentially a special file or database in which the system automatically keeps tack of all operations performed by users on the regular data. A typical trail might contain the following: request (source text), terminal from which the operation was invoked; user who invoked the operation; date and time of the operation; table(s), tuple(s), attribute(s) affected; old values and new values.
Mandatory access control

The classification levels adopted in this approach are assumed to form a strict ordering consisting of, e.g., \{top secret, secret, confidential\}. Similarly, we can assign such a clearance level to the users. Thus, the following simple rules are imposed:

1. User $i$ can retrieve object $j$ only if the clearance level of $i$ is greater than or equal to the classification level of $j$ (the “simple security property”);

2. User $i$ can update object $j$ only if the clearance level of $i$ is equal to the classification level of $j$ (the “star property”).
Why the second rule?

The second rule can be stated alternatively as *the file written by a user will automatically be given the same classification level as the clearance level of that user.*

The second rule is necessary to prevent a user with “top secret” status from copying data of “secret status” to a file with only “confidential” status so that this file can be read by a user with a lesser privilege. On the other hand, if we do want this update to happen, then the classification level of the file to be updated should be automatically promoted first.

This approach received lots of attention since DoD started to require any DB system it purchases supports this security measure.
Multi-level security

Assume that we want to apply the ideas of mandatory access control to the supplier table $S$, and we wish to control access to individual tuples. Then each tuple needs to be labeled with its classification level, e.g., $4=\text{top secret}$, $3=\text{secret}$, $2=\text{confidential}$, etc.. For example,

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
<th>CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
<td>2</td>
</tr>
<tr>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
<td>3</td>
</tr>
<tr>
<td>S3</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
<td>2</td>
</tr>
<tr>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
<td>4</td>
</tr>
<tr>
<td>S5</td>
<td>Adams</td>
<td>30</td>
<td>Athens</td>
<td>3</td>
</tr>
</tbody>
</table>

Now suppose users U1 and U2 have clearance levels 3, and 2, respectively. Then U1 and U2 will see table $S$ differently! A request to retrieve all suppliers will return four tuples: those for S1, S2, S3, and S5, if it is issued by U1, but just two tuples for S1 and S3, if issued by U2. Neither of them sees that for S4.
If we think about this case in terms of request modification, and consider the query $S$ WHERE CITY='London’, then surely, it will be automatically converted to $S$ WHERE CITY='London’ AND CLASS<=$user clearance.

This is an example of a *multi-level table*, and the fact that the same data looks different to different users is called *polyinstantiation*.

Other operations such as INSERT, UPDATE, and DELETE, are treated similarly.

**Homework:** Read through the rest of §17.3 *Mandatory access control.*
Statistical databases

This kind of database permits queries that derive aggregated information, e.g., sums, averages, but not queries that derive individual information. E.g., the query “What is the average salary of programmers?” might be permitted, while “What is Mary’s salary?” will not be.

The problem with such databases is that it might be possible to make inferences from legal queries to deduce the answers to illegal ones. Such problems will become more significant as the use of data warehouses increases.
### An example

<table>
<thead>
<tr>
<th>NAME</th>
<th>SEX</th>
<th>C#</th>
<th>OCCUP.</th>
<th>SAL.</th>
<th>TAX</th>
<th>AU.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alf</td>
<td>M</td>
<td>3</td>
<td>Prog.</td>
<td>50K</td>
<td>10K</td>
<td>3</td>
</tr>
<tr>
<td>Bea</td>
<td>F</td>
<td>2</td>
<td>Phys.</td>
<td>130K</td>
<td>10K</td>
<td>0</td>
</tr>
<tr>
<td>Cary</td>
<td>F</td>
<td>0</td>
<td>Prog.</td>
<td>56K</td>
<td>18K</td>
<td>1</td>
</tr>
<tr>
<td>Dawn</td>
<td>F</td>
<td>2</td>
<td>Builder</td>
<td>60K</td>
<td>12K</td>
<td>1</td>
</tr>
<tr>
<td>Ed</td>
<td>M</td>
<td>1</td>
<td>Clerk</td>
<td>44K</td>
<td>4K</td>
<td>0</td>
</tr>
<tr>
<td>Fay</td>
<td>F</td>
<td>1</td>
<td>Artist.</td>
<td>30K</td>
<td>0K</td>
<td>0</td>
</tr>
<tr>
<td>Guy</td>
<td>M</td>
<td>0</td>
<td>Lawy.</td>
<td>190K</td>
<td>0K</td>
<td>0</td>
</tr>
<tr>
<td>Hal</td>
<td>M</td>
<td>3</td>
<td>Homer</td>
<td>44K2</td>
<td>2K</td>
<td>0</td>
</tr>
<tr>
<td>Ivy</td>
<td>F</td>
<td>4</td>
<td>Prog.</td>
<td>64K</td>
<td>10K</td>
<td>1</td>
</tr>
<tr>
<td>Joy</td>
<td>F</td>
<td>1</td>
<td>Prog.</td>
<td>60K</td>
<td>20K</td>
<td>1</td>
</tr>
</tbody>
</table>

Assume that a user U is authorized to perform statistical queries only, but he wants to find out Alf’s salary.
Assume that U knows from outside sources that Alf is a programmer and is male, consider the following two queries:

\[
\text{WITH (STATS WHERE SEX='M' AND OCCUPATION='Programmer') AS X: COUNT (X)}
\]

Result 1.

\[
\text{WITH (STATS WHERE SEX='M' AND OCCUPATION='Programmer') AS X: SUM(X, SALARY)}
\]

Result 50K.

The security of the database has then been compromised, even though U issued only legitimate statistical queries. This example suggests that the system should refuse to respond to a query for which the cardinality of the set to be summarized is less than some lower bound \( b \), as well as when it is greater than the upper bound \( N - b \).
Since otherwise, we can also obtain Alf’s salary as follows:

COUNT (STATS)

Result: 12

WITH (STATS WHERE NOT (SEX='M' AND (OCCUPATION='Programmer')) AS X:
    COUNT (X)

Result: 11;

SUM (STATS, SALARY)

Result: 728K.

WITH (STATS WHERE NOT (SEX='M' AND (OCCUPATION='Programmer')) AS X:
    SUM (X, SALARY)

Result: 678K;
Nothing is unbeatable!

It can be shown that simply restricting queries to those for which the set to be summarized has cardinality \( c, b \leq c \leq N - b \), is inadequate to avoid compromise. An example of \( b=2 \) is given in pp. 515.

In general, for almost any statistical database, we can construct a boolean expression to find the answer to any inadmissible query, called a *general tracker*.

Hence, there is always a way to get in, no matter how many “locks” do you place on its “door”. If that happens, what should we do as a backup plan?
Data encryption

Data encryption is the most effective countermeasure against any attempt to bypass the system. Let’s call the original data the plain-text, which is encrypted by using an encryption algorithm. Such an algorithm accepts the plain text and an encryption key, and sends out the ciphertext. The algorithm might be made public, but never the key (?)..

For example, let’s assume that we only have to deal with uppercase letters and blanks, denoted as ‘+’, and consider the following example. Let the plain text be AS KINGFISHERS CATCH FIRE, and the key be ELIOT. Then an encryption algorithm might work as follows:
1. Divide the plaintext into blocks of length equal to that of the key: AS+KI, NGFIS, HERS+, CATCH, +FIRE.

2. Replace each character of the plaintext with integers in the range 00-26, using blank=00, A=01, ..., Z=26: 0119001109, 1407060919, 0805181900, 0301200308, 0006091805.

3. Repeat step 2 for the key: 0512091520.

4. For each block of the plaintext, replace each character with the sum, modulo 27, of its integer encoding and the integer encoding of the corresponding character of the key: 0604092602, 1919152412, 1317000720, 0813021801, 0518180625.

5. Replace each integer encoding in the result of step 4 by its character equivalent: FEIZB, SSOXL, MQ+GT, HMBRA, ERRFY

**Homework:** Decrypt the above ciphertext, and complete Exercise 17.5.
**Question:** How difficult is it for a would-be infiltrator to determine the key w/o prior knowledge, given matching plaintext and ciphertext?

**Answer:** Not very, only $7,893,600$ ways to guess the key. But, much more sophisticated schemes can easily be devised.

The above example follows a *substitution* procedure: an encryption key was used to determine, for each character of the plaintext, a ciphertext character to be substituted for that character. The decryption follows the opposite process.
An alternative

Another approach is a permutation one, in which plaintext characters are simply rearranged into some different sequence. For example, given the plaintext “HELLO”, and its 7 bit ASCII representation

1001000 1000101 1001100 1001100 1001111,

we can decipher it by swap the two adjacent bits, to turn it into the following:

0110001 0001010 0110011 0001100 0110111

Then send it to the receiver, who will get “HELLO” back by reversing the process.

Neither of these two methods is particularly secure in itself, but, algorithms that combine the two can provide quite high a degree of security.
Symmetric key encryption

Data encryption standard (DES) is such a combined procedure, and was adopted on Nov. 23, 1976 by NSA. The DES has the property that the decryption algorithm is identical to the encryption algorithm. Thus, it is just yet another traditional symmetric cipher, in which the cipher comes with an encryption key, which is the same as the decryption key.

Hence, the person who wants to send an encrypted message to another one has to send over the key first, and securely. All involved have to try their best to hide the key from any body else.
Asymmetric key encryption

In an asymmetric key scheme, besides the encryption algorithm, there are also two keys: a public key, and a private key, for everybody involved. Both the algorithm and the public key are known (phone book, T-shirt, Tattoo, ...), while the private key is not, and is “very difficult” to be figured out.

Thus, when Bob wants to send over an encrypted message to Dave, Bob simply finds out the public key of Dave, use it with the algorithm to encrypt the message, send it over to Dave. Dave will then use his private key to decrypt it.

If anybody else gets the encrypted message in any way, s/he will not be able to read it, since s/he “effectively” can’t decrypt it.
Symmetric vs. asymmetric

In a symmetric encryption, the sender of an encrypted message has to tell the receiver the encryption algorithm, together with the key, in a secure way. If during this process, there is any leak, then the whole process becomes meaningless.

In an asymmetric encryption, the only secret, the private key, is never sent. On the other hand, the non-secret part, the algorithm, together with the public key, are distributed as open as possible. Thus, the whole process is completely open, while the secret is kept entirely.
The well-known RSA approach implements the above asymmetric encryption mechanism. The name comes from its creators: Ronald Rivest, Adi Shamir, and Leonard Adleman. When using it, the encryptor uses a pair of two prime numbers as the private key, and publish their product as the public key.

RSA works because 1) There is a known fast algorithm to determine if a given number is prime; and 2) there is no known fast algorithm for finding the prime factor of a given composite number.

A specific example

RSA was first announced in 1977, when an encrypted text, together with the following public key, was printed in *Scientific American*:

\[ N = 114,381,625,757,888,867, 669, 235,779, 976,146,612,010, 218,296,721,242, 362,562, 561,842, 935,245,733,897,830,597, 123,563, 958,705, 058,989,075,147,599,290, 026,879, 543,541. \]

On April 26, 1994, a team of 600 volunteers found out the factors. In other words, it took 17 years to factor this number in the order of \(10^{123}\).

Today, for important banking transactions, \(N\) is typically in the order of \(10^{300}\). It would take more than 1,000 years to crack it, using a hundred million PCs.