17 SECURITY

I know that's a secret, for it's whispered everywhere.

—*William Congreve*

Database management systems are increasingly being used to store information about all aspects of an enterprise. The data stored in a DBMS is often vital to the business interests of the organization and is regarded as a corporate asset. In addition to protecting the intrinsic value of the data, corporations must consider ways to ensure privacy and to control access to data that must not be revealed to certain groups of users for various reasons.

In this chapter we discuss the concepts underlying access control and security in a DBMS. After introducing database security issues in Section 17.1, we consider two distinct approaches, called *discretionary* and *mandatory*, to specifying and managing access controls. An **access control** mechanism is a way to control the data that is accessible to a given user. After introducing access controls in Section 17.2 we cover discretionary access control, which is supported in SQL-92, in Section 17.3. We briefly cover mandatory access control, which is not supported in SQL-92, in Section 17.4.

In Section 17.5 we discuss several additional aspects of security, such as security in a statistical database, the role of the database administrator, and the use of techniques such as encryption and audit trails.

17.1 INTRODUCTION TO DATABASE SECURITY

There are three main objectives to consider while designing a secure database application:

1. **Secrecy:** Information should not be disclosed to unauthorized users. For example, a student should not be allowed to examine other students' grades.
2. **Integrity:** Only authorized users should be allowed to modify data. For example, students may be allowed to see their grades, yet not allowed (obviously!) to modify them.
3. **Availability:** Authorized users should not be denied access. For example, an instructor who wishes to change a grade should be allowed to do so.

To achieve these objectives, a clear and consistent **security policy** should be developed to describe what security measures must be enforced. In particular, we must determine what part of the data is to be protected and which users get access to which portions of the data. Next, the **security mechanisms** of the underlying DBMS (and OS, as well as external mechanisms such as securing access to buildings and so on) must be utilized to enforce the policy. We emphasize that security measures must be taken at several levels. Security leaks in the operating system or network connections can circumvent database security mechanisms. For example, such leaks could allow an intruder to log on as the database administrator with all the attendant DBMS access rights! Human factors are another source of security leaks. For example, a user may choose a password that is easy to guess, or a user who is authorized to see sensitive data may misuse it. Such errors in fact account for a large percentage of security breaches. We will not discuss these aspects of security despite their importance because they are not specific to database management systems.

Views provide a valuable tool in enforcing security policies. The view mechanism can be used to create a 'window' on a collection of data that is appropriate for some group of users. Views allow us to limit access to sensitive data by providing access to a restricted version (defined through a view) of that data, rather than to the data itself.

We use the following schemas in our examples:

> Sailors(*sid:* `integer`, *sname:* `string`, *rating:* `integer`, *age:* `real`)
> Boats(*bid:* `integer`, *bname:* `string`, *color:* `string`)
> Reserves(*sname:* `string`, *bid:* `integer`, *day:* `dates`)

Notice that Reserves has been modified to use *sname*, rather than *sid*.

17.2 ACCESS CONTROL

A database for an enterprise contains a great deal of information and usually has several groups of users. Most users need to access only a small part of the database to carry out their tasks. Allowing users unrestricted access to all the data can be undesirable, and a DBMS should provide mechanisms to control access to data.

A DBMS offers two main approaches to access control. **Discretionary access control** is based on the concept of access rights, or **privileges**, and mechanisms for giving users such privileges. A privilege allows a user to access some data object in a certain manner (e.g., to read or to modify). A user who creates a database object such as a table or a view automatically gets all applicable privileges on that object. The DBMS subsequently keeps track of how these privileges are granted to other users, and possibly revoked, and ensures that at all times only users with the necessary privileges can access an object. SQL-92 supports discretionary access control through the `GRANT` and `REVOKE` commands. The `GRANT` command gives privileges to users, and the `REVOKE` command takes away privileges. We discuss discretionary access control in Section 17.3.

Discretionary access control mechanisms, while generally effective, have certain weaknesses. In particular, a devious unauthorized user can trick an authorized user into disclosing sensitive data. **Mandatory access control** is based on systemwide policies that cannot be changed by individual users. In this approach each database object is assigned a *security class*, each user is assigned *clearance* for a security class, and rules are imposed on reading and writing of database objects by users. The DBMS determines whether a given user can read or write a given object based on certain rules that involve the security level of the object and the clearance of the user. These rules seek to ensure that sensitive data can never be 'passed on' to a user without the necessary clearance. The SQL-92 standard does not include any support for mandatory access control. We discuss mandatory access control in Section 17.4.

17.3 DISCRETIONARY ACCESS CONTROL

SQL-92 supports discretionary access control through the `GRANT` and `REVOKE` commands. The `GRANT` command gives users privileges to base tables and views. The syntax of this command is as follows:

`GRANT` **privileges** `ON` **object** `TO` **users** [`WITH GRANT OPTION`]

For our purposes **object** is either a base table or a view. SQL recognizes certain other kinds of objects, but we will not discuss them. Several privileges can be specified, including these:

- `SELECT`: The right to access (read) all columns of the table specified as the **object**, *including columns added later* through `ALTER TABLE` commands.
- `INSERT`(*column-name*): The right to insert rows with (non-*null* or nondefault) values in the named column of the table named as **object**. If this right is to be granted with respect to all columns, including columns that might be added later, we can simply use `INSERT`. The privileges `UPDATE`(*column-name*) and `UPDATE` are similar.
- `DELETE`: The right to delete rows from the table named as **object**.
- `REFERENCES`(*column-name*): The right to define foreign keys (in other tables) that refer to the specified column of the table **object**. `REFERENCES` without a column name specified denotes this right with respect to all columns, including any that are added later.

If a user has a privilege with the **grant option**, he or she can pass it to another user (with or without the grant option) by using the GRANT command. A user who creates a base table automatically has all applicable privileges on it, along with the right to grant these privileges to other users. A user who creates a view has precisely those privileges on the view that he or she has on *every* one of the view or base tables used to define the view. The user creating the view must have the SELECT privilege on each underlying table, of course, and so is always granted the SELECT privilege on the view. The creator of the view has the SELECT privilege with the grant option only if he or she has the SELECT privilege with the grant option on every underlying table. In addition, if the view is updatable and the user holds INSERT, DELETE, or UPDATE privileges (with or without the grant option) on the (single) underlying table, the user automatically gets the same privileges on the view.

Only the owner of a schema can execute the data definition statements CREATE, ALTER, and DROP on that schema. The right to execute these statements cannot be granted or revoked.

In conjunction with the GRANT and REVOKE commands, views are an important component of the security mechanisms provided by a relational DBMS. By defining views on the base tables, we can present needed information to a user while *hiding* other information that the user should not be given access to. For example, consider the following view definition:

```
CREATE VIEW ActiveSailors (name, age, day)
       AS SELECT S.sname, S.age, R.day
          FROM   Sailors S, Reserves R
          WHERE  S.sname = R.sname AND S.rating > 6
```

A user who can access ActiveSailors, but not Sailors or Reserves, knows which sailors have reservations but cannot find out the *bid*s of boats reserved by a given sailor.

Privileges are assigned in SQL-92 to **authorization ids**, which can denote a single user or a group of users; a user must specify an authorization id and, in many systems, a corresponding *password* before the DBMS accepts any commands from him or her. So, technically, *Joe*, *Michael*, and so on are authorization ids rather than user names in the following examples.

Suppose that user Joe has created the tables Boats, Reserves, and Sailors. Some examples of the GRANT command that Joe can now execute are listed below:

```
GRANT INSERT, DELETE ON Reserves TO Yuppy WITH GRANT OPTION
GRANT SELECT ON Reserves TO Michael
GRANT SELECT ON Sailors TO Michael WITH GRANT OPTION
GRANT UPDATE (rating) ON Sailors TO Leah
```

Role-based authorization in SQL: Privileges are assigned to users (authorization ids, to be precise) in SQL-92. In the real world, privileges are often associated with a user's job or *role* within the organization. Many DBMSs have long supported the concept of a **role** and allowed privileges to be assigned to roles. Roles can then be granted to users and other roles. (Of courses, privileges can also be granted directly to users.) The SQL:1999 standard includes support for roles. What is the benefit of including a feature that many systems already support? This ensures that over time, all vendors who comply with the standard will support this feature. Thus, users can use the feature without worrying about portability of their application across DBMSs.

```
GRANT REFERENCES (bid) ON Boats TO Bill
```

Yuppy can insert or delete Reserves rows and can authorize someone else to do the same. Michael can execute `SELECT` queries on Sailors and Reserves, and he can pass this privilege to others for Sailors, but not for Reserves. With the `SELECT` privilege, Michael can create a view that accesses the Sailors and Reserves tables (for example, the ActiveSailors view) but he cannot grant `SELECT` on ActiveSailors to others.

On the other hand, suppose that Michael creates the following view:

```
CREATE VIEW YoungSailors (sid, age, rating)
       AS SELECT S.sid, S.age, S.rating
          FROM   Sailors S
          WHERE  S.age < 18
```

The only underlying table is Sailors, for which Michael has `SELECT` with the grant option. He therefore has `SELECT` with the grant option on YoungSailors and can pass on the `SELECT` privilege on YoungSailors to Eric and Guppy:

```
GRANT SELECT ON YoungSailors TO Eric, Guppy
```

Eric and Guppy can now execute `SELECT` queries on the view YoungSailors—note, however, that Eric and Guppy do *not* have the right to execute `SELECT` queries directly on the underlying Sailors table.

Michael can also define constraints based on the information in the Sailors and Reserves tables. For example, Michael can define the following table, which has an associated table constraint:

```
CREATE TABLE Sneaky (maxrating   INTEGER,
                     CHECK ( maxrating >=
                                        ( SELECT MAX (S.rating )
                                          FROM   Sailors S )))
```

By repeatedly inserting rows with gradually increasing *maxrating* values into the Sneaky table until an insertion finally succeeds, Michael can find out the highest *rating* value in the Sailors table! This example illustrates why SQL requires the creator of a table constraint that refers to Sailors to possess the SELECT privilege on Sailors.

Returning to the privileges granted by Joe, Leah can update only the *rating* column of Sailors rows. She can execute the following command, which sets all ratings to 8:

```
UPDATE Sailors S
SET    S.rating = 8
```

However, she cannot execute the same command if the SET clause is changed to be SET *S.age = 25*, because she is not allowed to update the *age* field. A more subtle point is illustrated by the following command, which decrements the rating of all sailors:

```
UPDATE Sailors S
SET    S.rating = S.rating - 1
```

Leah cannot execute this command because it requires the SELECT privilege on the *S.rating* column and Leah does not have this privilege!

Bill can refer to the *bid* column of Boats as a foreign key in another table. For example, Bill can create the Reserves table through the following command:

```
CREATE TABLE Reserves ( sname CHAR(10) NOTNULL,
                        bid   INTEGER,
                        day   DATE,
                        PRIMARY KEY (bid, day),
                        UNIQUE (sname),
                        FOREIGN KEY (bid) REFERENCES Boats )
```

If Bill did not have the REFERENCES privilege on the *bid* column of Boats, he would not be able to execute this CREATE statement because the FOREIGN KEY clause requires this privilege.

Specifying just the INSERT (similarly, REFERENCES etc.) privilege in a GRANT command is not the same as specifying SELECT(*column-name*) for each column currently in the table. Consider the following command over the Sailors table, which has columns *sid*, *sname*, *rating*, and *age*:

```
GRANT INSERT ON Sailors TO Michael
```

Suppose that this command is executed and then a column is added to the Sailors table (by executing an ALTER TABLE command). Note that Michael has the INSERT privilege with respect to the newly added column! If we had executed the following GRANT command, instead of the previous one, Michael would not have the INSERT privilege on the new column:

```
GRANT  INSERT ON Sailors(sid), Sailors(sname), Sailors(rating),
       Sailors(age), TO Michael
```

There is a complementary command to GRANT that allows the withdrawal of privileges. The syntax of the REVOKE command is as follows:

```
REVOKE [ GRANT OPTION FOR ] privileges
       ON object FROM users { RESTRICT | CASCADE }
```

The command can be used to revoke either a privilege or just the grant option on a privilege (by using the optional GRANT OPTION FOR clause). One of the two alternatives, RESTRICT or CASCADE, must be specified; we will see what this choice means shortly.

The intuition behind the GRANT command is clear: The creator of a base table or a view is given all the appropriate privileges with respect to it and is allowed to pass these privileges—including the right to pass along a privilege!—to other users. The REVOKE command is, as expected, intended to achieve the reverse: A user who has granted a privilege to another user may change his mind and want to withdraw the granted privilege. The intuition behind exactly what effect a REVOKE command has is complicated by the fact that a user may be granted the same privilege multiple times, possibly by different users.

When a user executes a REVOKE command with the CASCADE keyword, the effect is to withdraw the named privileges or grant option from all users who currently hold these privileges *solely* through a GRANT command that was previously executed by the same user who is now executing the REVOKE command. If these users received the privileges with the grant option and passed it along, those recipients will also lose their privileges as a consequence of the REVOKE command unless they also received these privileges independently.

We illustrate the REVOKE command through several examples. First, consider what happens after the following sequence of commands, where Joe is the creator of Sailors.

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION     (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION     (executed by Art)
REVOKE SELECT ON Sailors FROM Art CASCADE            (executed by Joe)
```

Art loses the `SELECT` privilege on Sailors, of course. Then Bob, who received this privilege from Art, and only Art, also loses this privilege. Bob's privilege is said to be **abandoned** when the privilege that it was derived from (Art's `SELECT` privilege with grant option, in this example) is revoked. When the `CASCADE` keyword is specified, all abandoned privileges are also revoked (possibly causing privileges held by other users to become abandoned and thereby revoked recursively). If the `RESTRICT` keyword is specified in the `REVOKE` command, the command is rejected if revoking the privileges *just* from the users specified in the command would result in other privileges becoming abandoned.

Consider the following sequence, as another example:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION        (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION        (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION        (executed by Art)
REVOKE SELECT ON Sailors FROM Art CASCADE               (executed by Joe)
```

As before, Art loses the `SELECT` privilege on Sailors. But what about Bob? Bob received this privilege from Art, but he also received it independently (coincidentally, directly from Joe). Thus Bob retains this privilege. Consider a third example:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION        (executed by Joe)
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION        (executed by Joe)
REVOKE SELECT ON Sailors FROM Art CASCADE               (executed by Joe)
```

Since Joe granted the privilege to Art twice and only revoked it once, does Art get to keep the privilege? As per the SQL-92 standard, no. Even if Joe absentmindedly granted the same privilege to Art several times, he can revoke it with a single `REVOKE` command.

It is possible to revoke just the grant option on a privilege:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION        (executed by Joe)
REVOKE GRANT OPTION FOR SELECT ON Sailors
       FROM Art CASCADE                                 (executed by Joe)
```

This command would leave Art with the `SELECT` privilege on Sailors, but Art no longer has the grant option on this privilege and therefore cannot pass it on to other users.

These examples bring out the intuition behind the `REVOKE` command, but they also highlight the complex interaction between `GRANT` and `REVOKE` commands. When a `GRANT` is executed, a **privilege descriptor** is added to a table of such descriptors maintained by the DBMS. The privilege descriptor specifies the following: the *grantor* of the privilege, the *grantee* who receives the privilege, the *granted privilege* (including the name of the object involved), and whether the grant option is included. When a user creates a table or view and 'automatically' gets certain privileges, a privilege descriptor with *system* as the grantor is entered into this table.

The effect of a series of GRANT commands can be described in terms of an **authorization graph** in which the nodes are users—technically, they are authorization ids—and the arcs indicate how privileges are passed. There is an arc from (the node for) user 1 to user 2 if user 1 executed a GRANT command giving a privilege to user 2; the arc is labeled with the descriptor for the GRANT command. A GRANT command has no effect if the same privileges have already been granted to the same grantee by the same grantor. The following sequence of commands illustrates the semantics of GRANT and REVOKE commands when there is a *cycle* in the authorization graph:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION     (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION     (executed by Art)
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION     (executed by Bob)
GRANT SELECT ON Sailors TO Cal WITH GRANT OPTION     (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION     (executed by Cal)
REVOKE SELECT ON Sailors FROM Art CASCADE            (executed by Joe)
```

The authorization graph for this example is shown in Figure 17.1. Note that we indicate how Joe, the creator of Sailors, acquired the SELECT privilege from the DBMS by introducing a *System* node and drawing an arc from this node to Joe's node.

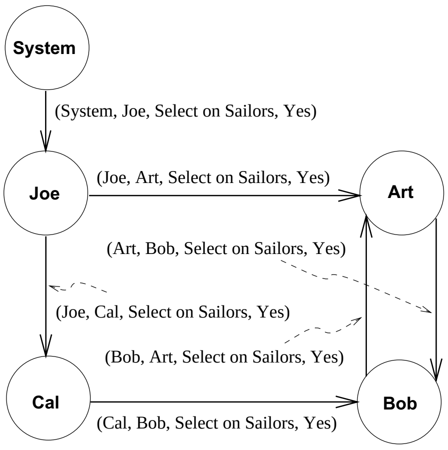
System
(System, Joe, Select on Sailors, Yes)
Joe
(Joe, Art, Select on Sailors, Yes)
Art
(Art, Bob, Select on Sailors, Yes)
(Joe, Cal, Select on Sailors, Yes)
(Bob, Art, Select on Sailors, Yes)
Cal
Bob
(Cal, Bob, Select on Sailors, Yes)

Figure 17.1 Example Authorization Graph

As the graph clearly indicates, Bob's grant to Art and Art's grant to Bob (of the same privilege) creates a cycle. Bob is subsequently given the same privilege by Cal, who received it independently from Joe. At this point Joe decides to revoke the privilege that he granted to Art.

Let us trace the effect of this revocation. The arc from Joe to Art is removed because it corresponds to the granting action that is revoked. All remaining nodes have the following property: *If node N has an outgoing arc labeled with a privilege, there is a path from the System node to node N in which each arc label contains the same privilege plus the grant option.* That is, any remaining granting action is justified by a privilege received (directly or indirectly) from the System. The execution of Joe's `REVOKE` command therefore stops at this point, with everyone continuing to hold the `SELECT` privilege on Sailors.

This result may seem unintuitive because Art continues to have the privilege only because he received it from Bob, and at the time that Bob granted the privilege to Art, he had received it only from Art! Although Bob acquired the privilege through Cal subsequently, shouldn't the effect of his grant to Art be undone when executing Joe's `REVOKE` command? The effect of the grant from Bob to Art is *not* undone in SQL-92. In effect, if a user acquires a privilege multiple times from different grantors, SQL-92 treats each of these grants to the user as having occurred *before* that user passed on the privilege to other users. This implementation of `REVOKE` is convenient in many real-world situations. For example, if a manager is fired after passing on some privileges to subordinates (who may in turn have passed the privileges to others), we can ensure that only the manager's privileges are removed by first redoing all of the manager's granting actions and then revoking his or her privileges. That is, we need not recursively redo the subordinates' granting actions.

To return to the saga of Joe and his friends, let us suppose that Joe decides to revoke Cal's `SELECT` privilege as well. Clearly, the arc from Joe to Cal corresponding to the grant of this privilege is removed. The arc from Cal to Bob is removed as well, since there is no longer a path from System to Cal that gives Cal the right to pass the `SELECT` privilege on Sailors to Bob. The authorization graph at this intermediate point is shown in Figure 17.2.

The graph now contains two nodes (Art and Bob) for which there are outgoing arcs with labels containing the `SELECT` privilege on Sailors; thus, these users have granted this privilege. However, although each node contains an incoming arc carrying the same privilege, *there is no such path from System to either of these nodes*; thus, these users' right to grant the privilege has been abandoned. We therefore remove the outgoing arcs as well. In general, these nodes might have other arcs incident upon them, but in this example, they now have no incident arcs. Joe is left as the only user with the `SELECT` privilege on Sailors; Art and Bob have lost their privileges.

17.3.1 Grant and Revoke on Views and Integrity Constraints *

The privileges held by the creator of a view (with respect to the view) change over time as he or she gains or loses privileges on the underlying tables. If the creator loses

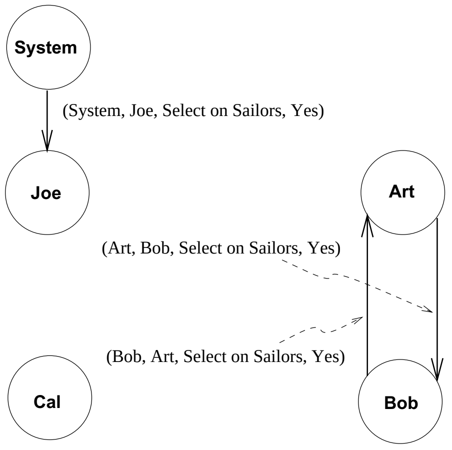
System
(System, Joe, Select on Sailors, Yes)
Joe
Art
(Art, Bob, Select on Sailors, Yes)
(Bob, Art, Select on Sailors, Yes)
Cal
Bob

Figure 17.2 Example Authorization Graph during Revocation

a privilege held with the grant option, users who were given that privilege on the view will lose it as well. There are some subtle aspects to the `GRANT` and `REVOKE` commands when they involve views or integrity constraints. We will consider some examples that highlight the following important points:

1. A view may be dropped because a `SELECT` privilege is revoked from the user who created the view.
2. If the creator of a view gains additional privileges on the underlying tables, he or she automatically gains additional privileges on the view.
3. The distinction between the `REFERENCES` and `SELECT` privileges is important.

Suppose that Joe created Sailors and gave Michael the `SELECT` privilege on it with the grant option, and Michael then created the view YoungSailors and gave Eric the `SELECT` privilege on YoungSailors. Eric now defines a view called FineYoungSailors:

```
CREATE VIEW FineYoungSailors (name, age, rating)
       AS SELECT S.sname, S.age, S.rating
          FROM   YoungSailors S
          WHERE  S.rating > 6
```

What happens if Joe revokes the `SELECT` privilege on Sailors from Michael? Michael no longer has the authority to execute the query used to define YoungSailors because the definition refers to Sailors. Therefore, the view YoungSailors is dropped (i.e., destroyed). In turn, FineYoungSailors is dropped as well. Both these view definitions are removed from the system catalogs; even if a remorseful Joe decides to give back the SELECT privilege on Sailors to Michael, the views are gone and must be created afresh if they are required.

On a more happy note, suppose that everything proceeds as described above until Eric defines FineYoungSailors; then, instead of revoking the SELECT privilege on Sailors from Michael, Joe decides to also give Michael the INSERT privilege on Sailors. Michael's privileges on the view YoungSailors are upgraded to what he would have if he were to create the view *now.* Thus he acquires the INSERT privilege on YoungSailors as well. (Note that this view is updatable.) What about Eric? His privileges are unchanged.

Whether or not Michael has the INSERT privilege on YoungSailors with the grant option depends on whether or not Joe gives him the INSERT privilege on Sailors with the grant option. To understand this situation, consider Eric again. If Michael has the INSERT privilege on YoungSailors with the grant option, he can pass this privilege to Eric. Eric could then insert rows into the Sailors table because inserts on YoungSailors are effected by modifying the underlying base table, Sailors. Clearly, we don't want Michael to be able to authorize Eric to make such changes unless Michael has the INSERT privilege on Sailors with the grant option.

The REFERENCES privilege is very different from the SELECT privilege, as the following example illustrates. Suppose that Joe is the creator of Boats. He can authorize another user, say Fred, to create Reserves with a foreign key that refers to the *bid* column of Boats by giving Fred the REFERENCES privilege with respect to this column. On the other hand, if Fred has the SELECT privilege on the *bid* column of Boats but not the REFERENCES privilege, Fred *cannot* create Reserves with a foreign key that refers to Boats. If Fred creates Reserves with a foreign key column that refers to *bid* in Boats, and later loses the REFERENCES privilege on the *bid* column of boats, the foreign key constraint in Reserves is dropped; however, the Reserves table is *not* dropped.

To understand why the SQL-92 standard chose to introduce the REFERENCES privilege, rather than to simply allow the SELECT privilege to be used in this situation, consider what happens if the definition of Reserves specified the NO ACTION option with the foreign key—Joe, the owner of Boats, may be prevented from deleting a row from Boats because a row in Reserves refers to this Boats row! Giving Fred, the creator of Reserves, the right to constrain updates on Boats in this manner goes beyond simply allowing him to read the values in Boats, which is all that the SELECT privilege authorizes.

17.4 MANDATORY ACCESS CONTROL *

Discretionary access control mechanisms, while generally effective, have certain weaknesses. In particular they are susceptible to *Trojan horse* schemes whereby a devious unauthorized user can trick an authorized user into disclosing sensitive data. For example, suppose that student Tricky Dick wants to break into the grade tables of instructor Trustin Justin. Dick does the following:

- He creates a new table called MineAllMine and gives `INSERT` privileges on this table to Justin (who is blissfully unaware of all this attention, of course).
- He modifies the code of some DBMS application that Justin uses often to do a couple of additional things: first, read the Grades table, and next, write the result into MineAllMine.

Then he sits back and waits for the grades to be copied into MineAllMine and later undoes the modifications to the application to ensure that Justin does not somehow find out later that he has been cheated. Thus, despite the DBMS enforcing all discretionary access controls—only Justin's authorized code was allowed to access Grades—sensitive data is disclosed to an intruder. The fact that Dick could surreptitiously modify Justin's code is outside the scope of the DBMS's access control mechanism.

Mandatory access control mechanisms are aimed at addressing such loopholes in discretionary access control. The popular model for mandatory access control, called the Bell-LaPadula model, is described in terms of **objects** (e.g., tables, views, rows, columns), **subjects** (e.g., users, programs), **security classes**, and **clearances**. Each database object is assigned a *security class*, and each subject is assigned *clearance* for a security class; we will denote the class of an object or subject A as *class(A)*. The security classes in a system are organized according to a partial order, with a **most secure class** and a **least secure class**. For simplicity, we will assume that there are four classes: *top secret (TS), secret (S), confidential (C), and unclassified (U)*. In this system, $TS > S > C > U$, where $A > B$ means that class A data is more sensitive than class B data.

The Bell-LaPadula model imposes two restrictions on all reads and writes of database objects:

1. **Simple Security Property:** Subject S is allowed to read object O only if *class(S)* $\geq$ *class(O)*. For example, a user with TS clearance can read a table with C clearance, but a user with C clearance is not allowed to read a table with TS classification.
2. ***-Property:** Subject S is allowed to write object O only if *class(S)* $\leq$ *class(O)*. For example, a user with S clearance can only write objects with S or TS classification.

If discretionary access controls are also specified, these rules represent additional restrictions. Thus, to read or write a database object, a user must have the necessary privileges (obtained via `GRANT` commands) *and* the security classes of the user and the object must satisfy the preceding restrictions. Let us consider how such a mandatory control mechanism might have foiled Tricky Dick. The Grades table could be classified as *S*, Justin could be given clearance for *S*, and Tricky Dick could be given a lower clearance (*C*). Dick can only create objects of *C* or lower classification; thus, the table MineAllMine can have at most the classification *C*. When the application program running on behalf of Justin (and therefore with clearance *S*) tries to copy Grades into MineAllMine, it is not allowed to do so because *class(MineAllMine)* < *class(application)*, and the *-Property is violated.

17.4.1 Multilevel Relations and Polyinstantiation

To apply mandatory access control policies in a relational DBMS, a security class must be assigned to each database object. The objects can be at the granularity of tables, rows, or even individual column values. Let us assume that each row is assigned a security class. This situation leads to the concept of a **multilevel table**, which is a table with the surprising property that users with different security clearances will see a different collection of rows when they access the same table.

Consider the instance of the Boats table shown in Figure 17.3. Users with *S* and *TS* clearance will get both rows in the answer when they ask to see all rows in Boats. A user with *C* clearance will get only the second row, and a user with *U* clearance will get no rows.

bid	*bname*	*color*	**Security Class**
101	Salsa	Red	*S*
102	Pinto	Brown	*C*

Figure 17.3 An Instance *B1* of Boats

The Boats table is defined to have *bid* as the primary key. Suppose that a user with clearance *C* wishes to enter the row ⟨*101,Picante,Scarlet,C*⟩. We have a dilemma:

- If the insertion is permitted, two distinct rows in the table will have key 101.
- If the insertion is not permitted because the primary key constraint is violated, the user trying to insert the new row, who has clearance *C*, can infer that there is a boat with *bid=101* whose security class is higher than *C*. This situation compromises the principle that users should not be able to infer any information about objects that have a higher security classification.

This dilemma is resolved by effectively treating the security classification as part of the key. Thus, the insertion is allowed to continue, and the table instance is modified as shown in Figure 17.4.

bid	*bname*	*color*	**Security Class**
101	Salsa	Red	*S*
101	Picante	Scarlet	*C*
102	Pinto	Brown	*C*

Figure 17.4 Instance *B1* after Insertion

Users with clearance *C* or *U* see just the rows for Picante and Pinto, but users with clearance *S* or *TS* see all three rows. The two rows with *bid=101* can be interpreted in one of two ways: only the row with the higher classification (Salsa, with classification *S*) actually exists, or both exist and their presence is revealed to users according to their clearance level. The choice of interpretation is up to application developers and users.

The presence of data objects that appear to have different values to users with different clearances (for example, the boat with *bid* 101) is called **polyinstantiation**. If we consider security classifications associated with individual columns, the intuition underlying polyinstantiation can be generalized in a straightforward manner, but some additional details must be addressed. We remark that the main drawback of mandatory access control schemes is their rigidity; policies are set by system administrators, and the classification mechanisms are not flexible enough. A satisfactory combination of discretionary and mandatory access controls is yet to be achieved.

17.4.2 Covert Channels, DoD Security Levels

Even if a DBMS enforces the mandatory access control scheme discussed above, information can flow from a higher classification level to a lower classification level through indirect means, called **covert channels**. For example, if a transaction accesses data at more than one site in a distributed DBMS, the actions at the two sites must be coordinated. The process at one site may have a lower clearance (say *C*) than the process at another site (say *S*), and both processes have to agree to commit before the transaction can be committed. This requirement can be exploited to pass information with an *S* classification to the process with a *C* clearance: The transaction is repeatedly invoked, and the process with the *C* clearance always agrees to commit, whereas the process with the *S* clearance agrees to commit if it wants to transmit a 1 bit and does not agree if it wants to transmit a 0 bit.

In this (admittedly tortuous) manner, information with an *S* clearance can be sent to a process with a *C* clearance as a stream of bits. This covert channel is an indirect violation of the intent behind the *-Property. Additional examples of covert channels can be found readily in statistical databases, which we discuss in Section 17.5.2.

Current systems: Commercial RDBMSs are available that support discretionary controls at the *C2* level and mandatory controls at the *B1* level. IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all support SQL-92's features for discretionary access control. In general, they do not support mandatory access control; Oracle does offer a version of their product with support for mandatory access control.

DBMS vendors have recently started implementing mandatory access control mechanisms (although they are not part of the SQL-92 standard) because the United States Department of Defense (DoD) requires such support for its systems. The DoD requirements can be described in terms of **security levels** *A, B, C*, and *D* of which *A* is the most secure and *D* is the least secure.

Level *C* requires support for discretionary access control. It is divided into sublevels *C1* and *C2*; *C2* also requires some degree of accountability through procedures such as login verification and audit trails. Level *B* requires support for mandatory access control. It is subdivided into levels *B1*, *B2*, and *B3*. Level *B2* additionally requires the identification and elimination of covert channels. Level *B3* additionally requires maintenance of audit trails and the designation of a **security administrator** (usually, but not necessarily, the DBA). Level *A*, the most secure level, requires a mathematical proof that the security mechanism enforces the security policy!

17.5 ADDITIONAL ISSUES RELATED TO SECURITY *

Security is a broad topic, and our coverage is necessarily limited. This section briefly touches on some additional important issues.

17.5.1 Role of the Database Administrator

The database administrator (DBA) plays an important role in enforcing the security-related aspects of a database design. In conjunction with the owners of the data, the DBA will probably also contribute to developing a security policy. The DBA has a special account, which we will call the **system account**, and is responsible for the overall security of the system. In particular the DBA deals with the following:

1. **Creating new accounts:** Each new user or group of users must be assigned an authorization id and a password. Note that application programs that access the database have the same authorization id as the user executing the program.
2. **Mandatory control issues:** If the DBMS supports mandatory control—some customized systems for applications with very high security requirements (for

example, military data) provide such support—the DBA must assign security classes to each database object and assign security clearances to each authorization id in accordance with the chosen security policy.

The DBA is also responsible for maintaining the **audit trail**, which is essentially the log of updates with the authorization id (of the user who is executing the transaction) added to each log entry. This log is just a minor extension of the log mechanism used to recover from crashes. Additionally, the DBA may choose to maintain a log of *all* actions, including reads, performed by a user. Analyzing such histories of how the DBMS was accessed can help prevent security violations by identifying suspicious patterns before an intruder finally succeeds in breaking in, or it can help track down an intruder after a violation has been detected.

17.5.2 Security in Statistical Databases

A **statistical database** is one that contains specific information on individuals or events but is intended to permit only statistical queries. For example, if we maintained a statistical database of information about sailors, we would allow statistical queries about average ratings, maximum age, and so on, but would not want to allow queries about individual sailors. Security in such databases poses some new problems because it is possible to **infer** protected information (such as an individual sailor's rating) from answers to permitted statistical queries. Such inference opportunities represent covert channels that can compromise the security policy of the database.

Suppose that sailor Sneaky Pete wants to know the rating of Admiral Horntooter, the esteemed chairman of the sailing club, and happens to know that Horntooter is the oldest sailor in the club. Pete repeatedly asks queries of the form "How many sailors are there whose age is greater than X?" for various values of X, until the answer is 1. Obviously, this sailor is Horntooter, the oldest sailor. Note that each of these queries is a valid statistical query and is permitted. Let the value of X at this point be, say, 65. Pete now asks the query, "What is the maximum rating of all sailors whose age is greater than 65?" Again, this query is permitted because it is a statistical query. However, the answer to this query reveals Horntooter's rating to Pete, and the security policy of the database is violated.

One approach to preventing such violations is to require that each query must involve at least some minimum number, say N, of rows. With a reasonable choice of N, Pete would not be able to isolate the information about Horntooter, because the query about the maximum rating would fail. This restriction, however, is easy to overcome. By repeatedly asking queries of the form, "How many sailors are there whose age is greater than X?" until the system rejects one such query, Pete identifies a set of N sailors, including Horntooter. Let the value of X at this point be 55. Now, Pete can ask two queries:

- "What is the sum of the ratings of all sailors whose age is greater than 55?" Since N sailors have age greater than 55, this query is permitted.

- "What is the sum of the ratings of all sailors, other than Horntooter, whose age is greater than 55, and sailor Pete?" Since the set of sailors whose ratings are added up now includes Pete instead of Horntooter, but is otherwise the same, the number of sailors involved is still N, and this query is also permitted.

From the answers to these two queries, say A_1 and A_2, Pete, who knows his rating, can easily calculate Horntooter's rating as $A_1 - A_2 + \textit{Pete's rating}$.

Pete succeeded because he was able to ask two queries that involved many of the same sailors. The number of rows examined in common by two queries is called their **intersection**. If a limit were to be placed on the amount of intersection permitted between any two queries issued by the same user, Pete could be foiled. Actually, a truly fiendish (and patient) user can generally find out information about specific individuals even if the system places a minimum number of rows bound (N) and a maximum intersection bound (M) on queries, but the number of queries required to do this grows in proportion to N/M. We can try to additionally limit the total number of queries that a user is allowed to ask, but two users could still conspire to breach security. By maintaining a log of all activity (including read-only accesses), such query patterns can be detected, hopefully before a security violation occurs. This discussion should make it clear, however, that security in statistical databases is difficult to enforce.

17.5.3 Encryption

A DBMS can use *encryption* to protect information in certain situations where the normal security mechanisms of the DBMS are not adequate. For example, an intruder may steal tapes containing some data or tap a communication line. By storing and transmitting data in an encrypted form, the DBMS ensures that such stolen data is not intelligible to the intruder.

The basic idea behind encryption is to apply an **encryption algorithm**, which may be accessible to the intruder, to the original data and a user-specified or DBA-specified **encryption key**, which is kept secret. The output of the algorithm is the encrypted version of the data. There is also a **decryption algorithm**, which takes the encrypted data and the encryption key as input and then returns the original data. Without the correct encryption key, the decryption algorithm produces gibberish. This approach forms the basis for the **Data Encryption Standard** (DES), which has been in use since 1977, with an encryption algorithm that consists of character substitutions and permutations. The main weakness of this approach is that authorized users must be told the encryption key, and the mechanism for communicating this information is vulnerable to clever intruders.

Another approach to encryption, called **public-key encryption**, has become increasingly popular in recent years. The encryption scheme proposed by Rivest, Shamir, and Adleman, called RSA, is a well-known example of public-key encryption. Each authorized user has a **public encryption key**, known to everyone, and a *private* **decryption key** (used by the decryption algorithm), chosen by the user and known only to him or her. The encryption and decryption algorithms themselves are assumed to be publicly known. Consider a user called Sam. Anyone can send Sam a secret message by encrypting the message using Sam's publicly known encryption key. Only Sam can decrypt this secret message because the decryption algorithm requires Sam's decryption key, known only to Sam. Since users choose their own decryption keys, the weakness of DES is avoided.

The main issue for public-key encryption is how encryption and decryption keys are chosen. Technically, public-key encryption algorithms rely on the existence of **one-way functions**, which are functions whose inverse is computationally very hard to determine. The RSA algorithm, for example, is based on the observation that although checking whether a given number is prime is easy, determining the prime factors of a nonprime number is extremely hard. (Determining the prime factors of a number with over 100 digits can take years of CPU-time on the fastest available computers today.)

We now sketch the intuition behind the RSA algorithm, assuming that the data to be encrypted is an integer I. To choose an encryption key and a decryption key, our friend Sam would first choose a very large integer *limit*, which we assume is larger than the largest integer that he will ever need to encode. Sam chooses *limit* to be the product of two (large!) distinct prime numbers, say $p * q$. Sam then chooses some prime number e, chosen to be larger than both p and q, as his encryption key. Both *limit* and e are made public and are used by the encryption algorithm.

Now comes the clever part: Sam chooses the decryption key d in a special way based on p, q, and e.[1] The essential point of the scheme is that it is easy to compute d given e, p, and q, but *very* hard to compute d given just e and *limit*. In turn, this difficulty depends on the fact that it is hard to determine the prime factors of *limit*, which happen to be p and q.

A very important property of the encryption and decryption algorithms in this scheme is that given the corresponding encryption and decryption keys, the algorithms are inverses of each other—not only can data be encrypted and then decrypted, but we can also apply the decryption algorithm first and then the encryption algorithm and still get the original data back! This property can be exploited by two users, say Elmer and Sam, to exchange messages in such a way that if Elmer gets a message that is supposedly from Sam, he can verify that it is from Sam (in addition to being able to decrypt the message), and further, *prove* that it is from Sam. This feature has obvious

[1]In case you are curious, d is chosen such that $d * e = 1 \textbf{ mod } ((p-1)*(q-1))$.

practical value. For example, suppose that Elmer's company accepts orders for its products over the Internet and stores these orders in a DBMS. The requirements are:

1. Only the company (Elmer) should be able to understand an order. A customer (say Sam) who orders jewelry frequently may want to keep the orders private (perhaps because he does not want to become a popular attraction for burglars!).

2. The company should be able to verify that an order that supposedly was placed by customer Sam was indeed placed by Sam, and not by an intruder claiming to be Sam. By the same token, Sam should not be able to claim that the company forged an order from him—an order from Sam must *provably* come from Sam.

The company asks each customer to choose an encryption key (Sam chooses e_{Sam}) and a decryption key (d_{Sam}) and to make the encryption key public. It also makes its own encryption key (e_{Elmer}) public. The company's decryption key (d_{Elmer}) is kept secret, and customers are expected to keep their decryption keys secret as well.

Now let's see how the two requirements can be met. To place an order, Sam could just encrypt the order using encryption key e_{Elmer}, and Elmer could decrypt this using decryption key d_{Elmer}. This simple approach satisfies the first requirement because d_{Elmer} is known only to Elmer. However, since e_{Elmer} is known to everyone, someone who wishes to play a prank could easily place an order on behalf of Sam without informing Sam. From the order itself, there is no way for Elmer to verify that it came from Sam. (Of course, one way to handle this is to give each customer an account and to rely on the login procedure to verify the identity of the user placing the order—the user would have to know the password for Sam's account—but the company may have thousands of customers and may not want to give each of them an account.)

A clever use of the encryption scheme, however, allows Elmer to verify whether the order was indeed placed by Sam. Instead of encrypting the order using e_{Elmer}, Sam first applies his *decryption algorithm*, using d_{Sam}, known only to Sam (and not even to Elmer!), to the original order. Since the order was not encrypted first, this produces gibberish, but as we shall see, there is a method in this madness. Next, Sam encrypts the result of the previous step using e_{Elmer} and registers the result in the database.

When Elmer examines such an order, he first decrypts it using d_{Elmer}. This step yields the gibberish that Sam generated from his order, because the encryption and decryption algorithm are inverses when applied with the right keys. Next, Elmer applies the *encryption algorithm* to this gibberish, using Sam's encryption key e_{Sam}, which is known to Elmer (and is public). This step yields the original unencrypted order, again because the encryption and decryption algorithm are inverses!

If the order had been forged, the forger could not have known Sam's decryption key d_{Sam}; the final result would have been nonsensical, rather than the original order.

Further, because the company does not know d_{Sam}, Sam cannot claim that a genuine order was forged by the company.

The use of public-key cryptography is not limited to database systems, but it is likely to find increasing application in the DBMS context thanks to the use of the DBMS as a repository for the records of sensitive commercial transactions. Internet commerce, as in the example above, could be a driving force in this respect.

17.6 POINTS TO REVIEW

- There are three main security objectives. First, information should not be disclosed to unauthorized users (*secrecy*). Second, only authorized users should be allowed to modify data (*integrity*). Third, authorized users should not be denied access (*availability*). A *security policy* describes the security measures enforced. These measures use the *security mechanisms* of the underlying DBMS. **(Section 17.1)**

- There are two main approaches to enforcing security measures. In *discretionary access control*, users have *privileges* to access or modify objects in the database. If they have permission, users can grant their privileges to other users, and the DBMS keeps track of who has what rights. In *mandatory access control*, objects are assigned security classes. Users have security clearance for a security class. Rules involving the security class and a user's clearance determine which database objects the user can access. **(Section 17.2)**

- SQL supports discretionary access through the `GRANT` and `REVOKE` commands. The creator of a table has automatically all privileges on it and can pass privileges on to other users or revoke privileges from other users. The effect of `GRANT` commands can be described as adding edges into an *authorization graph* and the effect of `REVOKE` commands can be described as removing edges from the graph. **(Section 17.3)**

- In mandatory access control, objects are organized into several *security classes* and users are organized into several levels of *clearance*. The security classes form a partial order. Reads and writes of an object are restricted by rules that involve the security class of the object and the clearance of the user. Users with different levels of clearance might see different records in the same table. This phenomenon is called *polyinstantiation*. **(Section 17.4)**

- The database administrator is responsible for the overall security of the system. The DBA has a *system account* with special privileges. The DBA also maintains an *audit trail*, a log of accesses to the DBMS with the corresponding user identifiers. *Statistical databases* only allow summary queries, but clever users can infer information about specific individuals from the answers to valid statistical queries.

We can use *encryption* techniques to ensure that stolen data cannot be deciphered. **(Section 17.5)**

EXERCISES

Exercise 17.1 Briefly answer the following questions based on this schema:

> Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`)
> Works(*eid:* `integer`, *did:* `integer`, *pct_time:* `integer`)
> Dept(*did:* `integer`, *budget:* `real`, *managerid:* `integer`)

1. Suppose you have a view SeniorEmp defined as follows:

```
CREATE VIEW SeniorEmp (sname, sage, salary)
       AS SELECT E.ename, E.age, E.salary
          FROM   Emp E
          WHERE  E.age > 50
```

   Explain what the system will do to process the following query:

```
SELECT S.sname
FROM   SeniorEmp S
WHERE  S.salary > 100,000
```

2. Give an example of a view on Emp that could be automatically updated by updating Emp.
3. Give an example of a view on Emp that would be impossible to update (automatically) and explain why your example presents the update problem that it does.
4. Consider the following view definition:

```
CREATE VIEW DInfo (did, manager, numemps, totsals)
       AS SELECT   D.did, D.managerid, COUNT (*), SUM (E.salary)
          FROM     Emp E, Works W, Dept D
          WHERE    E.eid = W.eid AND W.did = D.did
          GROUP BY D.did, D.managerid
```

   (a) Give an example of a view update on DInfo that could (in principle) be implemented automatically by updating one or more of the relations Emp, Works, and Dept. Does SQL-92 allow such a view update?

   (b) Give an example of a view update on DInfo that cannot (even in principle) be implemented automatically by updating one or more of the relations Emp, Works, and Dept. Explain why.

   (c) How could the view DInfo help in enforcing security?

Exercise 17.2 You are the DBA for the VeryFine Toy Company, and you create a relation called Employees with fields *ename*, *dept*, and *salary*. For authorization reasons, you also define views EmployeeNames (with *ename* as the only attribute) and DeptInfo with fields *dept* and *avgsalary*. The latter lists the average salary for each department.

1. Show the view definition statements for EmployeeNames and DeptInfo.
2. What privileges should be granted to a user who needs to know only average department salaries for the Toy and CS departments?
3. You want to authorize your secretary to fire people (you'll probably tell him whom to fire, but you want to be able to delegate this task), to check on who is an employee, and to check on average department salaries. What privileges should you grant?
4. Continuing with the preceding scenario, you don't want your secretary to be able to look at the salaries of individuals. Does your answer to the previous question ensure this? Be specific: Can your secretary possibly find out salaries of *some* individuals (depending on the actual set of tuples), or can your secretary always find out the salary of any individual that he wants to?
5. You want to give your secretary the authority to allow other people to read the EmployeeNames view. Show the appropriate command.
6. Your secretary defines two new views using the EmployeeNames view. The first is called AtoRNames and simply selects names that begin with a letter in the range A to R. The second is called HowManyNames and counts the number of names. You are so pleased with this achievement that you decide to give your secretary the right to insert tuples into the EmployeeNames view. Show the appropriate command, and describe what privileges your secretary has after this command is executed.
7. Your secretary allows Todd to read the EmployeeNames relation and later quits. You then revoke the secretary's privileges. What happens to Todd's privileges?
8. Give an example of a view update on the above schema that cannot be implemented through updates to Employees.
9. You decide to go on an extended vacation, and to make sure that emergencies can be handled, you want to authorize your boss Joe to read and modify the Employees relation and the EmployeeNames relation (and Joe must be able to delegate authority, of course, since he's too far up the management hierarchy to actually do any work). Show the appropriate SQL statements. Can Joe read the DeptInfo view?
10. After returning from your (wonderful) vacation, you see a note from Joe, indicating that he authorized his secretary Mike to read the Employees relation. You want to revoke Mike's `SELECT` privilege on Employees, but you don't want to revoke the rights that you gave to Joe, even temporarily. Can you do this in SQL?
11. Later you realize that Joe has been quite busy. He has defined a view called AllNames using the view EmployeeNames, defined another relation called StaffNames that he has access to (but that you can't access), and given his secretary Mike the right to read from the AllNames view. Mike has passed this right on to his friend Susan. You decide that even at the cost of annoying Joe by revoking some of his privileges, you simply have to take away Mike and Susan's rights to see your data. What `REVOKE` statement would you execute? What rights does Joe have on Employees after this statement is executed? What views are dropped as a consequence?

Exercise 17.3 Briefly answer the following questions.

1. Explain the intuition behind the two rules in the Bell-LaPadula model for mandatory access control.

2. Give an example of how covert channels can be used to defeat the Bell-LaPadula model.
3. Give an example of polyinstantiation.
4. Describe a scenario in which mandatory access controls prevent a breach of security that cannot be prevented through discretionary controls.
5. Describe a scenario in which discretionary access controls are required to enforce a security policy that cannot be enforced using only mandatory controls.
6. If a DBMS already supports discretionary and mandatory access controls, is there a need for encryption?
7. Explain the need for each of the following limits in a statistical database system:
   (a) A maximum on the number of queries a user can pose.
   (b) A minimum on the number of tuples involved in answering a query.
   (c) A maximum on the intersection of two queries (i.e., on the number of tuples that both queries examine).
8. Explain the use of an audit trail, with special reference to a statistical database system.
9. What is the role of the DBA with respect to security?
10. What is public-key encryption? How does it differ from the encryption approach taken in the Data Encryption Standard (DES), and in what ways is it better than DES?
11. What are one-way functions, and what role do they play in public-key encryption?
12. Explain how a company offering services on the Internet could use public-key encryption to make its order-entry process secure. Describe how you would use DES encryption for the same purpose, and contrast the public-key and DES approaches.

PROJECT-BASED EXERCISES

Exercise 17.4 Is there any support for views or authorization in Minibase?

BIBLIOGRAPHIC NOTES

The authorization mechanism of System R, which greatly influenced the `GRANT` and `REVOKE` paradigm in SQL-92, is described in [290]. A good general treatment of security and cryptography is presented in [179], and an overview of database security can be found in [119] and [404]. Security in statistical databases is investigated in several papers, including [178] and [148]. Multilevel security is discussed in several papers, including [348, 434, 605, 621].